The Pragmatic Programmers

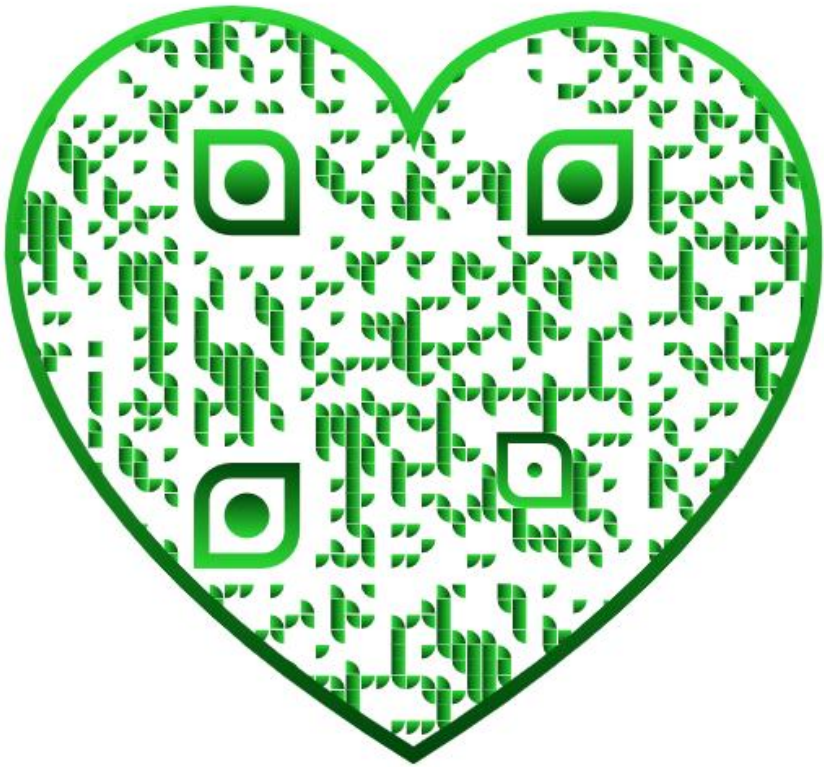
Programming Elixir ≥ 1.6

Functional
|> Concurrent
|> Pragmatic
|> Fun



Dave Thomas

Foreword by
José Valim,
Creator of Elixir



Programming Elixir ≥ 1.6

Functional |> Concurrent |> Pragmatic |> Fun

by Dave Thomas

Version: P1.0 (May 2018)

Copyright © 2018 The Pragmatic Programmers, LLC. This book is licensed to the individual who purchased it. We don't copy-protect it because that would limit your ability to use it for your own purposes. Please don't break this trust—you can use this across all of your devices but please do not share this copy with other members of your team, with friends, or via file sharing services. Thanks.

Many of the designations used by manufacturers and sellers to distinguish their products are

claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or

omissions, or for damages that may result from the use of information (including program listings) contained herein.

About the Pragmatic Bookshelf

The Pragmatic Bookshelf is an agile publishing company. We're here because we want to improve the lives of developers. We do this by creating timely, practical titles, written by programmers for programmers.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic

titles, please visit us at http://pragprog.com.

Our ebooks do not contain any Digital Restrictions Management, and have always been DRM-free. We pioneered the beta book concept, where you can purchase and read a book while it’s still being written, and provide feedback to the author to help make a better book for everyone. Free resources for all purchasers include source code downloads (if applicable), errata and discussion forums, all available on the book's home page at pragprog.com. We’re here to

make your life easier.

New Book Announcements

Want to keep up on our latest titles and announcements, and occasional special offers? Just create an account on pragprog.com (an email address and a password is all it takes) and select the checkbox to receive newsletters. You can also follow us on twitter as @pragprog.

About Ebook Formats

If you buy directly from pragprog.com, you get ebooks in all available formats for one price. You can synch your ebooks amongst all your devices

(including iPhone/iPad, Android, laptops, etc.) via Dropbox. You get free updates for the life of the edition. And, of course, you can always come back and re-download your books when needed. Ebooks bought from the Amazon Kindle store are subject to Amazon's polices. Limitations in Amazon's file format may cause ebooks to display differently on different devices. For more information, please see our FAQ at pragprog.com/frequently-asked-questions/ebooks. To learn more about this book and access the free resources, go to https://

pragprog.com/book/elixir16, the book's homepage.

Thanks for your continued support,

Andy Hunt
The Pragmatic Programmers

The team that produced this book includes:
Andy Hunt (Publisher)
Janet Furlow (VP of Operations)
Brian MacDonald (Managing Editor)
Jacquelyn Carter (Supervising Editor)
Candace Cunningham (Copy Editor)

Potomac Indexing, LLC
(Indexing)
Gilson Graphics (Layout)

For customer support, please
contact support@pragprog.com.

For international rights, please
contact rights@pragprog.com.

Table of Contents

1. **Foreword**

2. **A Vain Attempt at a Justification, Take Two**
    1. Acknowledgments

3. **1. Take the Red Pill**
    1. Programming Should Be About Transforming Data
    2. Installing Elixir
    3. Running Elixir
    4. Suggestions for Reading the Book
    5. Exercises
    6. Think Different(ly)

4. **Part I. Conventional Programming**

1. **2. Pattern Matching**

1. Assignment:
   I Do Not Think It Means What You Think It Means.
2. More Complex Matches
3. Ignoring a Value with _ (Underscore)
4. Variables Bind Once (per Match)
5. Another Way of Looking at the Equals Sign

2. **3.**

Immutability

1. You Already Have (Some) Immutable Data
2. Immutable Data Is Known Data
3. Performance Implications of Immutability
4. Coding with Immutable Data

3. **4.**

Elixir Basics

1. Built-in Types
2. Value Types
3. System Types
4. Collection Types
5. Maps
6. Binaries
7. Dates and Times
8. Names, Source Files, Conventions, Operators, and So On
9. Variable Scope
10. End of the Basics

4. **5.**

Anonymous Functions

1. Functions and Pattern Matching
2. One Function, Multiple Bodies
3. Functions Can Return Functions
4. Passing Functions as Arguments
5. Functions Are the Core

5. **6.**

Modules and Named Functions

1. Compiling a Module
2. The Function’s Body Is a Block
3. Function Calls and Pattern Matching
4. Guard Clauses
5. Default Parameters
6. Private Functions
7. The Amazing Pipe Operator: |>
8. Modules
9. Module Attributes
10. Module Names: Elixir, Erlang, and Atoms
11. Calling a Function in an Erlang Library
12. Finding Libraries

6. **7.**

Lists and Recursion

1. Heads and Tails

2. Using Head and Tail to Process a List
3. Using Head and Tail to Build a List
4. Creating a Map Function
5. Reducing a List to a Single Value
6. More Complex List Patterns
7. The List Module in Action
8. Get Friendly with Lists

7. **8.**
   Maps, Keyword Lists, Sets, and Structs
   1. How to Choose Between Maps, Structs, and Keyword Lists
   2. Keyword Lists
   3. Maps
   4. Pattern Matching and Updating Maps
   5. Updating a Map
   6. Structs
   7. Nested Dictionary Structures
   8. Sets
   9. With Great Power Comes Great Temptation

8. **9.**
   An Aside—What Are Types?

9. **10.**
   Processing Collections—Enum and Stream
   1. Enum—Processing Collections

2. Streams—Lazy Enumerables
3. The Collectable Protocol
4. Comprehensions
5. Moving Past Divinity

10. **11.**
Strings and Binaries
1. String Literals
2. The Name “strings”
3. Single-Quoted Strings—Lists of Character Codes
4. Binaries
5. Double-Quoted Strings Are Binaries
6. Binaries and Pattern Matching
7. Familiar Yet Strange

11. **12.**
Control Flow
1. if and unless
2. cond
3. case
4. Raising Exceptions
5. Designing with Exceptions
6. Doing More with Less

12. **13.**
Organizing a Project
1. The Project: Fetch Issues from GitHub

2. Step 1: Use Mix to Create Our New Project
3. Transformation: Parse the Command Line
4. Write Some Basic Tests
5. Refactor: Big Function Alert
6. Transformation: Fetch from GitHub
7. Step 2: Use Libraries
8. Transformation: Convert Response
9. Transformation: Sort Data
10. Transformation: Take First *n* Items
11. Transformation: Format the Table
12. Step 3: Make a Command-Line Executable
13. Step 4: Add Some Logging
14. Step 5: Create Project Documentation
15. Coding by Transforming Data

13. **14. Tooling**

1. Debugging with IEx
2. Testing
3. Code Dependencies
4. Server Monitoring
5. Source-Code Formatting
6. Inevitably, There's More

5. **Part II. Concurrent Programming**

1. **15.**

   Working with Multiple Processes

   1. A Simple Process
   2. Process Overhead
   3. When Processes Die
   4. Parallel Map—The “Hello, World” of Erlang
   5. A Fibonacci Server
   6. Agents—A Teaser
   7. Thinking in Processes

2. **16.**

   Nodes—The Key to Distributing Services

   1. Naming Nodes
   2. Naming Your Processes
   3. Input, Output, PIDs, and Nodes
   4. Nodes Are the Basis of Distribution

3. **17.**

   OTP: Servers

   1. Some OTP Definitions
   2. An OTP Server
   3. GenServer Callbacks
   4. Naming a Process
   5. Tidying Up the Interface
   6. Making Our Server into a Component

4. **18.**

OTP: Supervisors

1. Supervisors and Workers
2. Worker Restart Options
3. Supervisors Are the Heart of Reliability

5. **19.**

A More Complex Example

1. Introduction to Duper
2. The Duper Application
3. But Does It Work?
4. Planning Your Elixir Application
5. Next Up

6. **20.**

OTP: Applications

1. This Is Not Your Father's Application
2. The Application Specification File
3. Turning Our Sequence Program into an OTP Application
4. Supervision Is the Basis of Reliability
5. Releasing Your Code
6. Distillery—The Elixir Release Manager
7. OTP Is Big—Unbelievably Big

7. **21.**

Tasks and Agents

1. Tasks

2. Agents
3. A Bigger Example
4. Agents and Tasks, or GenServer?

6. **Part III. More Advanced Elixir**

1. **22.**
Macros and Code Evaluation
1. Implementing an if Statement
2. Macros Inject Code
3. Using the Representation as Code
4. Using Bindings to Inject Values
5. Macros Are Hygienic
6. Other Ways to Run Code Fragments
7. Macros and Operators
8. Digging Deeper
9. Digging Ridiculously Deep

2. **23.**
Linking Modules: Behavio(u)rs and use
1. Behaviours
2. use and __using__
3. Putting It Together—Tracing Method Calls
4. Use use

3. **24.**

Protocols—Polymorphic Functions

1. Defining a Protocol
2. Implementing a Protocol
3. The Available Types
4. Protocols and Structs
5. Built-in Protocols
6. Protocols Are Polymorphism

4. **25.**

More Cool Stuff

1. Writing Your Own Sigils
2. Multi-app Umbrella Projects
3. But Wait! There's More!

7. **A1.**

Exceptions: raise and try, catch and throw

1. Raising an Exception
2. catch, exit, and throw
3. Defining Your Own Exceptions
4. Now Ignore This Appendix

8. **A2.**

Type Specifications and Type Checking

1. When Specifications Are Used
2. Specifying a Type
3. Defining New Types
4. Specs for Functions and Callbacks

5. Using Dialyzer

9.

Bibliography

Copyright © 2018, The Pragmatic Bookshelf.

Foreword

I have always been fascinated with how changes in hardware affect how we write software.

A couple of decades ago, memory was a very limited resource. It made sense back then for our software to take hold of some piece of memory and mutate it as necessary. However, allocating this memory and cleaning up after we no longer needed it was a very error-prone task. Some memory was never freed; sometimes memory was allocated over another structure, leading to faults. At the time, garbage collection was a

known technique, but we needed faster CPUs in order to use it in our daily software and free ourselves from manual memory management. That has happened—most of our languages are now garbage-collected.

Today, a similar phenomenon is happening. Our CPUs are not getting any faster. Instead, our computers get more and more cores. This means new software needs to use as many cores as it can if it is to maximize its use of the machine. This conflicts directly with how we currently write software.

In fact, mutating our memory state actually slows down our software when many cores are involved. If you have four cores trying to access and

manipulate the same piece of memory, they can trip over each other. This potentially corrupts memory unless some kind of synchronization is applied.

I quickly learned that applying this synchronization is manual, error prone, and tiresome, and it hurts performance. I suddenly realized that's not how I wanted to spend time writing software in the next years of my career, and I set out to study new languages and technologies.

It was on this quest that I fell in love with the Erlang virtual machine and ecosystem.

In the Erlang VM, all code runs in tiny concurrent processes, each with its own

state. Processes talk to each other via messages. And since all communication happens by message-passing, exchanging messages between different machines on the same network is handled transparently by the VM, making it a perfect environment for building distributed software!

However, I felt there was still a gap in the Erlang ecosystem. I missed first-class support for some of the features I find necessary in my daily work—things such as metaprogramming, polymorphism, and first-class tooling. From this need, Elixir was born.

Elixir is a pragmatic approach to functional programming. It values its functional foundations and it focuses on

developer productivity. Concurrency is the backbone of Elixir software. As garbage collection once freed developers from the shackles of memory management, Elixir is here to free you from antiquated concurrency mechanisms and bring you joy when writing concurrent code.

A functional programming language lets us think in terms of functions that transform data. This transformation never mutates data. Instead, each application of a function potentially creates a new, fresh version of the data. This greatly reduces the need for data-synchronization mechanisms.

Elixir also empowers developers by providing macros. Elixir code is nothing

more than data, and therefore can be manipulated via macros like any other value in the language.

Finally, object-oriented programmers will find many of the mechanisms they consider essential to writing good software, such as polymorphism, in Elixir.

All this is powered by the Erlang VM, a 20-year-old virtual machine built from scratch to support robust, concurrent, and distributed software. Elixir and the Erlang VM are going to change how you write software and make you ready to tackle the upcoming years in programming.

José Valim
Creator of Elixir

Tenczynek, Poland, October 2014

Copyright © 2018, The Pragmatic Bookshelf.

A Vain Attempt at a Justification, Take Two

I'm a language nut. I love trying languages out, and I love thinking about their design and implementation. (I know; it's sad.)

I came across Ruby in 1998 because I was an avid reader of comp.lang.misc (ask your parents). I downloaded it, compiled it, and fell in love. As with any time you fall in love, it's difficult to explain why. It just worked the way I work, and it had enough depth to keep me interested.

Fast-forward 15 years. All that time I'd been looking for something new that gave me the same feeling.

I came across Elixir a while back, but for some reason never got sucked in. But a few months before starting on the first edition of this book, I was chatting with Corey Haines. I was bemoaning the fact that I wanted a way to show people functional programming concepts without the academic trappings those books seem to attract. He told me to look again at Elixir. I did, and I felt the same way I felt when I first saw Ruby.

So now I'm dangerous. I want other people to see just how great this is. I want to evangelize. So I write a book. But I don't want to write another 900-

page Pickaxe book. I want this book to be short and exciting. So I'm not going into all the detail, listing all the syntax, all the library functions, all the OTP options, or….

Instead, I want to give you an idea of the power and beauty of this programming model. I want to inspire you to get involved, and then point to the online resources that will fill in the gaps.

But mostly, I want you to have fun.

Fast-forward three years. Elixir has moved on. Phoenix, its connectivity framework, introduced a whole new set of developers to the joys of a functional approach. The Nerves project makes it easy to write embedded Elixir code on

Linux-based microcontrollers. The Elixir base has grown—there are international, national, and regional conferences. Job ads ask for Elixir developers.

I've been moving on, too. But I'm still using Elixir daily. I just completed my second year as an adjunct professor at Southern Methodist University, corrupting the programmers of tomorrow with the temptations of Elixir. I've written an online Elixir course.[1]

And now I'm revving this book. To be honest, I don't really have to: Elixir 1.6 is not so different from 1.3 that the older book would not be useful. But my own thinking about Elixir has matured. I now do some things differently. And I'd like

to share these things with you.

Acknowledgments

It seems to be a common thread—the languages I fall in love with are created by people who are both clever and extremely nice. José Valim, the creator of Elixir, takes both of these adjectives to a new level. I owe him a massive thank-you for giving me so much fun over the last 18 months. Along with him, the whole Elixir core team has done an amazing job of cranking out an entire ecosystem that feels way more mature than its years. Thank you, all.

A conversation with Corey Haines reignited my interest in Elixir—thank you, Corey, for good evenings, some

interesting times in Bangalore, and the inspiration.

Bruce Tate is always an interesting sounding board, and his comments on early drafts of the book made a big difference. And I've been blessed with an incredible number of active and insightful beta readers who have made literally hundreds of suggestions for improvements. Thank you, all.

A big tip of the hat to Jessica Kerr, Anthony Eden, and Chad Fowler for letting me steal their tweets.

Kim Shrier seems to have been involved with my writing since before I started writing. Thanks, Kim, for another set of perceptive and detailed critiques.

Candace Cunningham again amazed me with her detailed copy editing: it's rare to find someone who can correct both your grammar and your code. The crew at Potomac did their customary stellar job of indexing.

Dave Thomas
mailto:dave@pragdave.me
Dallas, TX, April 2018

Footnotes

[1] https://codestool.coding-gnome.com

Copyright © 2018, The Pragmatic Bookshelf.

Chapter 1

Take the Red Pill

The Elixir programming language wraps functional programming with immutable state and an actor-based approach to concurrency in a tidy, modern syntax. And it runs on the industrial-strength, high-performance, distributed Erlang VM. But what does all that mean?

It means you can stop worrying about many of the difficult things that currently consume your time. You no longer have to think too hard about protecting your data consistency in a multithreaded environment. You worry

less about scaling your applications. And, most importantly, you can think about programming in a different way.

Programming Should Be About Transforming Data

If you come from an object-oriented world, then you are used to thinking in terms of classes and their instances. A class defines behavior, and objects hold state. Developers spend time coming up with intricate hierarchies of classes that try to model their problem, much as Victorian scientists created taxonomies of butterflies.

When we code with objects, we're thinking about state. Much of our time is spent calling methods in objects and passing them other objects. Based on these calls, objects update their own state, and possibly the state of other

objects. In this world, the class is king—it defines what each instance can do, and it implicitly controls the state of the data its instances hold. Our goal is data-hiding.

But that's not the real world. In the real world, we don't want to model abstract hierarchies (because in reality there aren't that many true hierarchies). We want to get things done, not maintain state.

Right now, for instance, I'm taking empty computer files and transforming them into files containing text. Soon I'll transform those files into a format you can read. A web server somewhere will transform your request to download the book into an HTTP response containing

the content.

I don’t want to hide data. I want to transform it.

Combine Transformations with Pipelines

Unix users are accustomed to the philosophy of small, focused command-line tools that can be combined in arbitrary ways. Each tool takes an input, transforms it, and writes the result in a format the next tool (or a human) can use.

This philosophy is incredibly flexible and leads to fantastic reuse. The Unix utilities can be combined in ways undreamed of by their authors. And each one multiplies the potential of the others.

It's also highly reliable—each small program does one thing well, which makes it easier to test.

There's another benefit. A command pipeline can operate in parallel. If I write

the word-count program, wc, runs at the same time as the grep command. Because wc consumes grep's output as it is produced, the answer is ready with virtually no delay once grep finishes.

Just to give you a taste of this, here's an Elixir function called pmap. It takes a collection and a function, and returns the list that results from applying that

function to each element of the collection. But…it runs a separate process to do the conversion of each element. Don't worry about the details for now.

spawn/pmap1.exs

We could run this function to get the squares of the numbers from 1 to 1,000.

And, yes, I just kicked off 1,000 background processes, and I used all the cores and processors on my machine.

The code may not make much sense, but by about halfway through the book, you'll be writing this kind of thing for

yourself.

Functions Are Data Transformers

Elixir lets us solve the problem in the same way the Unix shell does. Rather than have command-line utilities, we have functions. And we can string them together as we please. The smaller—more focused—those functions, the more flexibility we have when combining them.

If we want, we can make these functions run in parallel—Elixir has a simple but powerful mechanism for passing messages between them. And these are not your father's boring old processes or threads—we're talking about the potential to run millions of them on a single machine and have hundreds of

these machines interoperating. Bruce Tate commented on this paragraph with this thought: “Most programmers treat threads and processes as a necessary evil; Elixir developers feel they are an important simplification.” As we get deeper into the book, you’ll start to see what he means.

This idea of transformation lies at the heart of functional programming: a function transforms its inputs into its output. The trigonometric function *sin* is an example—give it π/4, and you’ll get back 0.7071. An HTML templating system is a function; it takes a template containing placeholders and a list of named values, and produces a completed HTML document.

But this power comes at a price. You're going to have to unlearn a whole lot of what you *know* about programming. Many of your instincts will be wrong. And this will be frustrating, because you're going to feel like a total n00b.

Personally, I feel that's part of the fun. You didn't learn, say, object-oriented programming overnight. You are unlikely to become a functional programming expert by breakfast, either.

But at some point things will click. You'll start thinking about problems in a different way, and you'll find yourself writing code that does amazing things with very little effort on your part. You'll find yourself writing small

chunks of code that can be used over and over, often in unexpected ways (just as wc and grep can be).

Your view of the world may even change a little as you stop thinking in terms of responsibilities and start thinking in terms of getting things done. And just about everyone can agree that will be fun.

Installing Elixir

This book assumes you're using at least Elixir 1.6. The most up-to-date instructions for installing Elixir are available at http://elixir-lang.org/install.html. Go install it now.

Running Elixir

In this book, I show a terminal session like this:

The terminal prompt is the dollar sign, and the stuff you type follows. (On your system, the prompt will likely be different.) Output from the system is shown without highlighting.

iex—Interactive Elixir

To test that your Elixir installation was successful, let’s start an interactive Elixir session. At your regular shell prompt, type iex.

(The various version numbers you see will likely be different—I won't bother to show them on subsequent examples.)

Once you have an IEx prompt, you can enter Elixir code and you'll see the result. If you enter an expression that continues over more than one line, IEx will prompt for the additional lines with an ellipsis (…).

The number in the prompt increments for each complete expression executed. I'll omit the number in most of the examples that follow.

There are several ways of exiting from IEx—none are tidy. The easiest two are

typing Ctrl-C twice or typing Ctrl-G followed by q and Return. On some systems, you can also use a single Ctrl-\.

IEx Helpers

IEx has a number of helper functions. Type h (followed by Return) to get a list:

In the list of helper functions, the number following the slash is the number of arguments the helper expects.

Probably the most useful is h itself. With an argument, it gives you help on Elixir modules or individual functions in a module. This works for any modules loaded into IEx (so when we talk about projects later on, you'll see your own documentation here, too). For example,

the IO module performs common input/output functions. For help on the module, type h(IO) or h IO:

This book frequently uses the puts function in the IO module, which in its simplest form writes a string to the console. Let's get the documentation:

Another informative helper is i, which displays information about a value:

IEx is a surprisingly powerful tool. Use it to compile and execute entire projects, log in to remote machines, and access running Elixir applications.

And, if you happen to include the occasional bug in your code (deliberately, of course), IEx has a simple debugger. We'll talk about it when we look at tooling.

Customizing iex

You can customize IEx by setting options. For example, I like showing the results of evaluations in bright cyan. To find out how to do that, I used this:

```
```

I then created a file called .iex.exs in my home directory, containing

```
```

If your IEx session looks messed up (and things such as [33m appear in the

output), it's likely your console does not support ANSI escape sequences. In that case, disable colorization using

You can put any Elixir code into .iex.exs.

Compile and Run

Once you tire of writing one-line programs in IEx, you'll want to start putting code into source files. These files will typically have the extension .ex or .exs. This is a convention—files ending in .ex are intended to be compiled into bytecodes and then run, whereas those ending in .exs are more like programs in scripting languages—they are effectively interpreted at the

source level. When we come to write tests for our Elixir programs, you'll see that the application files have .ex extensions, whereas the tests have .exs because we don't need to keep compiled versions of the tests lying around.

Let's write the classic first program. Go to a working directory and create a file called hello.exs.

intro/hello.exs

That example shows how most of the code listings in this book are presented. The bar before the code itself shows the path and file name that contains the code. If you're reading an ebook, you'll be able to click on this to download the

source file. You can also download all the code by visiting the book's page on our site and clicking on the *Source Code* link.[2]



Source file names are written in lowercase with underscores. They will have the extension .ex for programs that you intend to compile into binary form, and .exs for scripts that you want to run without compiling. Our "Hello, World" example is essentially throw-away code, so we used the .exs extension for it.

Having created our source file, let's run it. In the same directory where you created the file, run the elixir command:

We can also compile and run it inside IEx using the c helper:

The c helper compiled and executed the source file. The [] that follows the output is the return value of the c function—if the source file had contained any modules, their names would have been listed here.

The c helper compiled the source file as freestanding code. You can also load a file as if you'd typed each line into IEx using import_file. In this case, local variables set in the file are available in the IEx session.

As some folks fret over such things, the

Elixir convention is to use two-column indentation and spaces (not tabs).

Suggestions for Reading the Book

This book is not a top-to-bottom reference guide to Elixir. Instead, it is intended to give you enough information to know what questions to ask and when to ask them. So approach what follows with a spirit of adventure. Try the code as you read, and don't stop there. Ask yourself questions and then try to answer them, either by coding or by searching the web.

Participate in the book's discussion forums and consider joining the Elixir mailing list.[3]

You're joining the Elixir community

while it is still young. Things are exciting and dynamic, and there are plenty of opportunities to contribute.

Exercises

You'll find exercises sprinkled throughout this book. To become familiar with a language, you need to go beyond reading a book and following along with the examples; you need to write some code yourself. These exercises are starting points so you can do some exploring of the language. Try them out, and don't be afraid to make mistakes.

Think Different(ly)

This is a book about thinking differently—about accepting that some of the things folks say about programming may not be the full story:

- Object orientation is not the only way to design code.
- Functional programming need not be complex or mathematical.
- The bases of programming are not assignments, if statements, and loops.
- Concurrency does not need locks, semaphores, monitors, and the like.
- Processes are not necessarily expensive resources.
- Metaprogramming is not just something tacked onto a language.
- Even if it is work, programming should be fun.

Of course, I'm not saying Elixir is a magic potion (well, technically it is, but

you know what I mean). It isn't the *one true way* to write code. But it's different enough from the mainstream that learning it will give you more perspective and will open your mind to new ways of thinking about programming.

So let's start.

And remember to make it fun.

Footnotes

[2] http://pragprog.com/titles/elixir16

[3] https://elixirforum.com

Copyright © 2018, The Pragmatic Bookshelf.

Part 1
Conventional Programming

Elixir is great for writing highly parallel, reliable applications.

But to be a great language for parallel programming, a language first has to be great for conventional, sequential programming. In this part of the book we'll cover how to write Elixir code, and explore the idioms and conventions that make Elixir so powerful.

We cover:

- *How pattern matching binds values to variables*
- *How matching handles structured data*
- *How _ (underscore) lets you ignore a match*

Chapter 2

Pattern Matching

We started the previous chapter by saying Elixir engenders a different way of thinking about programming.

To illustrate this and to lay the

foundation for a lot of Elixir programming, let's start reprogramming your brain by looking at one of the cornerstones of all programming languages—assignment.

iex>

Assignment:
I Do Not Think It Means What You Think It Means.

Let's use the interactive Elixir shell, IEx, to look at a simple piece of code. (Remember, you start IEx at the command prompt using the `iex` command. You enter Elixir code at its `iex>` prompt, and it displays the resulting values.)

Most programmers would look at this code and say, "OK, we assign 1 to a variable `a`, then on the next line we add 3 to `a`, giving us 4."

But when it comes to Elixir, they'd be

wrong. In Elixir, the equals sign is not an assignment. Instead it's like an assertion. It succeeds if Elixir can find a way of making the left-hand side equal the right-hand side. Elixir calls the `=` symbol the *match operator*.

In this case, the left-hand side is a variable and the right-hand side is an integer literal, so Elixir can make the match true by binding the variable `a` to value 1. You could argue it *is* just an assignment. But let's take it up a notch.

Look at the second line of code, `1 = a`. It's another match, and it passes. The variable `a` already has the value 1 (it was set in the first line), so what's on the left

of the equals sign is the same as what's on the right, and the match succeeds.

But the third line, `2 = a`, raises an error. You might have expected it to assign 2 to `a`, as that would make the match succeed, but Elixir will only change the value of a variable on the left side of an equals sign—on the right a variable is replaced with its value. This failing line of code is the same as `2 = 1`, which causes the error.

More Complex Matches

First, a little background syntax. Elixir lists can be created using square brackets containing a comma-separated set of values. Here are some lists:

Back to the match operator.

To make the match true, Elixir bound the variable list to the list [1, 2, 3].

But let's try something else:

Elixir looks for a way to make the value

of the left side the same as the value of the right side. The left side is a list containing three variables, and the right is a list of three values, so the two sides could be made the same by setting the variables to the corresponding values.

Elixir calls this process *pattern matching*. A pattern (the left side) is matched if the values (the right side) have the same structure and if each term in the pattern can be matched to the corresponding term in the values. A literal value in the pattern matches that exact value, and a variable in the pattern matches by taking on the corresponding value.

Let’s look at a few more examples.

The value on the right side that corresponds to the term c on the left side is the sublist [3,4,5]; that is the value given to c to make the match true.

Let's try a pattern containing some values and variables.

The literal 2 in the pattern matched the corresponding term on the right, so the match succeeds by setting the values of a and b to 1 and 3. But…

Here the 1 (the second term in the list) cannot be matched against the corresponding element on the right side,

so no variables are set and the match fails. This gives us a way of matching a list that meets certain criteria—in this case a length of 3, with 1 as its second element.

Your Turn

- Exercise: PatternMatching-1

  Which of the following will match?

  - `a = [1, 2, 3]`
  - `a = 4`
  - `4 = a`
  - `[a, b] = [ 1, 2, 3 ]`
  - `a = [ [ 1, 2, 3 ] ]`
  - `[a] = [ [ 1, 2, 3 ] ]`
  - `[[a]] = [ [ 1, 2, 3 ] ]`

[illegible]

Ignoring a Value with _ (Underscore)

If we didn’t need to capture a value during the match, we could use the special variable _ (an underscore). This acts like a variable but immediately discards any value given to it—in a pattern match, it is like a wildcard saying, “I’ll accept any value here.” The following example matches any three-element list that has a 1 as its first element.

[illegible]hError)

Variables Bind Once (per Match)

Once a variable has been bound to a value in the matching process, it keeps that value for the remainder of the match.

The first expression in this example succeeds because a is initially matched with the first 1 on the right side. The value in a is then used in the second term to match the second 1 on the right side.

In the next expression, the first b matches the 1. But the second b corresponds to the value 2 on the right.

`b` cannot have two different values, and so the match fails.

However, a variable can be bound to a new value in a subsequent match, and its current value does not participate in the new match.

What if you instead want to force Elixir to use the existing value of the variable in the pattern? Prefix it with `^` (a caret). In Elixir, we call this the *pin operator*.

This also works if the variable is a component of a pattern:

There’s one more important part of pattern matching, which we’ll look at when we start digging deeper into lists.

Your Turn

- Exercise: PatternMatching-2

  Which of the following will match?

  - `[ a, b, a ] = [ 1, 2, 3 ]`
  - `[ a, b, a ] = [ 1, 1, 2 ]`
  - `[ a, b, a ] = [ 1, 2, 1 ]`

- Exercise: PatternMatching-3

  The variable `a` is bound to the value `2`. Which of the following will match?

  - `[ a, b, a ] = [ 1, 2, 3 ]`
  - `[ a, b, a ] = [ 1, 1, 2 ]`
  - `a = 1`
  - `^a = 2`
  - `^a = 1`
  - `^a = 2 - a`

Another Way of Looking at the Equals Sign

Elixir's pattern matching is similar to Erlang's (the main difference being that Elixir allows a match to reassign to a variable that was assigned in a prior match, whereas in Erlang a variable can be assigned only once).

Joe Armstrong, Erlang's creator, compares the equals sign in Erlang to that used in algebra. When you write the equation $x = a + 1$, you are not assigning the value of $a + 1$ to x. Instead you're simply asserting that the expressions x and $a + 1$ have the same value. If you know the value of x, you can work out the value of a, and vice

versa.

His point is that you had to unlearn the algebraic meaning of `=` when you first came across assignment in imperative programming languages. Now's the time to un-unlearn it.

That's why I talk about pattern matching as the first chapter in this part of the book. It is a core part of Elixir—we'll also use it in conditions, function calls, and function invocation.

But really, I wanted to get you thinking differently about programming languages and to show you that some of your existing assumptions won't work in Elixir.

And speaking of existing assumptions… the next chapter kills another sacred cow. Your current programming language is probably designed to make it easy to change data. After all, that's what programs do, right? Not Elixir. Let's talk about a language in which *all* data is immutable.

Copyright © 2018, The Pragmatic Bookshelf.

> *Change and decay in all around I see…*
>
> *Henry Francis Lyte "Abide with Me"*

Chapter 3

Immutability

If you listen to functional-programming aficionados, you'll hear people making a big deal about immutability—the fact that in a functional program, data cannot be altered once created.

And, indeed, Elixir enforces immutable

data.

But why?

Imagine programming in a world where you could not rely on that—where some other code, possibly running in parallel with your own, could change the value of 99. In that world, the call to do_something_with might kick off code that runs in the background, passing it the value 99 as an argument. And that could change the contents of the parameter it receives. Suddenly, 99 could be 100.

You'd be (rightly) upset. And, what's worse, you'd never really be able to guarantee your code produced the correct results.

Still thinking about your current language, consider this:

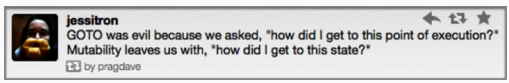
Again, you'd hope the print call would output [1,2,3]. But in most languages, do_something_with will receive the array as a reference. If it decides to change the second element or delete the contents entirely, the output will not be what you expect. Now it is harder to look at your code and reason about what it does.

Take this a step further—run multiple threads, all with access to the array. Who knows what state the array will be in if they all start changing it?

All this is because most compound data structures in most programming languages are mutable—you can change all or part of their content. And if pieces of your code do this in parallel, you're

in a world of hurt.

By coincidence, Jessica Kerr (@jessitron) tweeted the following on the day I updated this section:


jessitron
GOTO was evil because we asked, "how did I get to this point of execution?"
Mutability leaves us with, "how did I get to this state?"
by pragdave

It’s spot-on.

Immutable Data Is Known Data

Elixir sidesteps these problems. In Elixir, all values are immutable. The most complex nested list, the database record—these things behave just like the simplest integer. Their values are all immutable.

In Elixir, once a variable references a list such as [1,2,3], you know it will always reference those same values (until you rebind the variable). And this makes concurrency a lot less frightening.

But what if you *need* to add 100 to each element in [1,2,3]? Elixir does it by

producing a copy of the original, containing the new values. The original remains unchanged, and your operation will not affect any other code holding a reference to that original.

This fits in nicely with the idea that programming is about transforming data. When we update [1,2,3], we don't hack it in place. Instead we transform it into something new.

Performance Implications of Immutability

It would be easy to assume that this approach to programming is inefficient. After all, you have to create a new copy of data whenever you update it, and that's going to leave lots of old values around to be garbage-collected. Let's look at these in turn.

Copying Data

Although common sense might dictate that all this copying of data is inefficient, the reverse is true. Because Elixir knows that existing data is immutable, it can reuse it, in part or as a whole, when building new structures.

Consider the following code. (It uses a new operator, `[ head | tail ]`, which builds a new list with `head` as its first element and `tail` as the rest. We'll spend a whole chapter on this when we talk about lists and recursion. For now, just trust.)

In most languages, `list2` would be built by creating a new list containing the values 4, 3, 2, and 1. The three values in `list1` would be copied into the tail of `list2`. And that would be necessary because `list1` would be mutable.

But Elixir knows `list1` will never change, so it simply constructs a new list with a head of 4 and a tail of `list1`.

Garbage Collection

The other performance issue with a transformational language is that you quite often end up leaving old values unused when you create new values from them. This leaves a bunch of things using up memory on the heap, so garbage collection has to reclaim them.

Most modern languages have a garbage collector, and developers have grown to be suspicious of them—they can impact performance quite badly.

But the cool thing about Elixir is that you write your code using lots and lots of processes, and each process has its own heap. The data in your application is divvied up between these processes, so each individual heap is much, much

smaller than would have been the case if all the data had been in a single heap. As a result, garbage collection runs faster. If a process terminates before its heap becomes full, all its data is discarded—no garbage collection is required.

Coding with Immutable Data

Once you accept the concept, coding with immutable data is surprisingly easy. You just have to remember that any function that transforms data will return a new copy of it. Thus, we never capitalize a string. Instead, we return a capitalized copy of a string.

If you're coming from an object-oriented language, you may dislike that we write `String.capitalize name` and not `name.capitalize()`. But in object-oriented languages, objects mostly have mutable state. When you make a call such as `name.capitalize()` you have no

immediate indication whether you are changing the internal representation of the name, returning a capitalized copy, or both. There's plenty of scope for ambiguity.

In a functional language, we *always* transform data. We never modify it in place. The syntax reminds us of this every time we use it.

That's enough theory. It's time to start learning the language. In the next chapter we'll quickly go over the basic data types and some syntax, and in the following chapters we'll look at functions and modules.

Copyright © 2018, The Pragmatic Bookshelf.

We cover:

- *Five value types*
- *Two system types*
- *Four collection types*
- *Naming, operators, etc.*
- *The with expression*

Chapter 4

Elixir Basics

In this chapter we'll look at the types that are baked into Elixir, along with a few other things you need to know to get started. This chapter is deliberately terse—you're a programmer and you

know what an integer is, so I’m not going to insult you. Instead, I try to cover the Elixir-specific stuff you need to know.

Built-in Types

Elixir's built-in types are

- Value types:
    - Arbitrary-sized integers
    - Floating-point numbers
    - Atoms
    - Ranges
    - Regular expressions
- System types:
    - PIDs and ports
    - References
- Collection types:
    - Tuples
    - Lists
    - Maps
    - Binaries

Functions are a type too. They have their own chapter, following this one.

You might be surprised that this list doesn't include things such as strings and structures. Elixir has them, but they are built using the basic types from this list. However, they are important. Strings have their own chapter, and we have a couple of chapters on lists and maps (and other dictionary-like types). The maps chapter also describes the Elixir structure facilities.

Finally, there's some debate about whether regular expressions and ranges are value types. Technically they aren't —under the hood they are just structures. But right now it's convenient to treat them as distinct types.

Value Types

The value types in Elixir represent numbers, names, ranges, and regular expressions.

Integers

Integer literals can be written as decimal (`1234`), hexadecimal (`0xcafe`), octal (`0o765`), and binary (`0b1010`).

Decimal numbers may contain underscores—these are often used to separate groups of three digits when writing large numbers, so one million could be written `1_000_000` (or perhaps `100_0000` in China and Japan).

There is no fixed limit on the size of

integers—their internal representation grows to fit their magnitude.

(You'll see how to write a function such as factorial in Modules and Named Functions,.)

Floating-Point Numbers

Floating-point numbers are written using a decimal point. There must be at least one digit before and after the decimal point. An optional trailing exponent may be given. These are all valid floating-point literals:

Floats are IEEE 754 double precision, giving them about 16 digits of accuracy

and a maximum exponent of around 10^{308}.

Atoms

Atoms are constants that represent something's name. We write them using a leading colon (:), which can be followed by an atom word or an Elixir operator. An atom word is a sequence of UTF-8 letters (including combining marks), digits, underscores, and at signs (@). It may end with an exclamation point or a question mark. You can also create atoms containing arbitrary characters by enclosing the characters following the colon in double quotes. These are all atoms:

An atom's name is its value. Two atoms with the same name will always compare as being equal, even if they were created by different applications on two computers separated by an ocean.

We'll be using atoms a lot to tag values.

Ranges

Ranges are represented as *start*..*end*, where *start* and *end* are integers.

Regular Expressions

Elixir has regular-expression literals, written as ~r{regexp} or ~r{regexp} opts. Here I show the delimiters for regular-expression literals as { and }, but they are considerably more flexible. You can choose any nonalphanumeric characters as delimiters, as described in

the discussion of sigils. Some people use ~r/…/ for nostalgic reasons, but this is less convenient than the bracketed forms, as any forward slashes inside the pattern must be escaped.

Elixir regular expression support is provided by PCRE,[4] which basically provides a Perl 5–compatible syntax for patterns.

You can specify one or more single-character options following a regexp literal. These modify the literal's match behavior or add functionality.

You manipulate regular expressions with the Regex module.

System Types

These types reflect resources in the underlying Erlang VM.

PIDs and Ports

A PID is a reference to a local or remote process, and a port is a reference to a resource (typically external to the application) that you'll be reading or writing.

The PID of the current process is available by calling self. A new PID is created when you spawn a new process. We'll talk about this in Part II.

References

The function make_ref creates a globally

unique reference; no other reference will be equal to it. We don’t use references in this book.

Collection Types

The types we've seen so far are common in other programming languages. Now we're getting into more exotic types, so we'll go into more detail here.

Elixir collections can hold values of any type (including other collections).

Tuples

A tuple is an ordered collection of values. As with all Elixir data structures, once created a tuple cannot be modified.

You write a tuple between braces, separating the elements with commas.

A typical Elixir tuple has two to four elements—any more and you'll probably want to look at maps,, or structs,.

You can use tuples in pattern matching:

It is common for functions to return a tuple where the first element is the atom :ok if there were no errors. Here's an example (assuming you have a file called mix.exs in your current directory):

Because the file was successfully opened, the tuple contains an :ok status and a PID, which is how we access the

contents.

A common practice is to write matches that assume success:

The second open failed, and returned a tuple where the first element was :error. This caused the match to fail, and the error message shows that the second element contains the reason—enoent is Unix-speak for “file does not exist.”

Lists

We’ve already seen Elixir’s list literal syntax, [1,2,3]. This might lead you to think lists are like arrays in other languages, but they are not. (In fact, tuples are the closest Elixir gets to a

conventional array.) Instead, a list is effectively a linked data structure.

(If you've used the language Lisp, then this will all seem very familiar.)

As we'll discuss in Chapter 7, *Lists and Recursion*, this recursive definition of a list is the core of much Elixir programming.

Because of their implementation, lists are easy to traverse linearly, but they are expensive to access in random order. (To get to the nth element, you have to scan through n–1 previous elements.) It is always cheap to get the head of a list and to extract the tail of a list.

Lists have one other performance characteristic. Remember that we said all Elixir data structures are immutable? That means once a list has been made, it will never be changed. So, if we want to remove the head from a list, leaving just the tail, we never have to copy the list. Instead we can return a pointer to the tail. This is the basis of all the list-traversal tricks we'll cover in Chapter 7, *Lists and Recursion*.

Elixir has some operators that work specifically on lists:

Keyword Lists

Because we often need simple lists of key/value pairs, Elixir gives us a

shortcut. If we write

Elixir converts it into a list of two-value tuples:

Elixir allows us to leave off the square brackets if a keyword list is the last argument in a function call. Thus,

can be written more cleanly as

We can also leave off the brackets if a keyword list appears as the last item in any context where a list of values is expected.

[illegible]"}

Maps

A map is a collection of key/value pairs. A map literal looks like this:

Here are some maps:

In the first case the keys are strings, in the second they're tuples, and in the third they're atoms. Although typically all the keys in a map are the same type, that isn't required.

If the key is an atom, you can use the same shortcut that you use with

keyword lists:

You can also use expressions for the keys in map literals:

Why do we have both maps and keyword lists? Maps allow only one entry for a particular key, whereas keyword lists allow the key to be repeated. Maps are efficient (particularly as they grow), and they can be used in Elixir's pattern matching, which we discuss in later chapters.

In general, use keyword lists for things such as command-line parameters and passing around options, and use maps

when you want an associative array.

Accessing a Map

You extract values from a map using the key. The square-bracket syntax works with all maps:

If the keys are atoms, you can also use a dot notation:

You'll get a KeyError if there's no matching key when you use the dot notation.

Binaries

Sometimes you need to access data as a sequence of bits and bytes. For example, the headers in JPEG and MP3 files contain fields where a single byte may encode two or three separate values.

Elixir supports this with the binary data type. Binary literals are enclosed between `<<` and `>>`.

The basic syntax packs successive integers into bytes:

```
```

You can add modifiers to control the type and size of each individual field.

Here's a single byte that contains three fields of widths 2, 4, and 2 bits. (The example uses some built-in libraries to show the result's binary value.)

Binaries are both important and arcane. They're important because Elixir uses them to represent UTF strings. They're arcane because, at least initially, you're unlikely to use them directly.

[illegible](d1)

Dates and Times

Elixir 1.3 added a calendar module and four new date- and time-related types. Initially, they were little more than data holders, but Elixir 1.5 started to add some functionality to them.

The Calendar module represents the rules used to manipulate dates. The only current implementation is Calendar.ISO, the ISO-8601 representation of the Gregorian calendar.[5]

The Date type holds a year, a month, a day, and a reference to the ruling calendar.

(The sequences ~D[...] and ~T[...] are examples of Elixir's *sigils*. They are a way of constructing literal values. We'll see them again when we look at strings and binaries.)

Elixir can also represent a range of dates:

The Time type contains an hour, a minute, a second, and fractions of a second. The fraction is stored as a tuple containing microseconds and the number of significant digits. (The fact that time values track the number of significant digits in the seconds field means that ~T[12:34:56.0] is not equal to ~T[12:34:56.00].)

There are two date/time types: DateTime and NaiveDateTime. The naive version contains just a date and a time; DateTime adds the ability to associate a time zone. The ~N[...] sigil constructs NaiveDateTime structs.

If you are using dates and times in your code, you might want to augment these built-in types with a third-party library, such as Lau Taarnskov's Calendar library.[6]

[illegible]!

Names, Source Files, Conventions, Operators, and So On

Elixir identifiers must start with a letter or underscore, optionally followed by letters, digits, and underscores. Here *letter* means any UTF-8 *letter character* (optionally with a *combining mark*) and *digit* means a UTF-8 *decimal-digit character*. If you're using ASCII, this does what you'd expect. The identifiers may end with a question mark or an exclamation mark.

Here are some examples of valid variables:

And some examples of invalid

variables:

Module, record, protocol, and behavior names start with an uppercase letter and are BumpyCase. All other identifiers start with a lowercase letter or an underscore, and by convention use underscores between words. If the first character is an underscore, Elixir doesn't report a warning if the variable is unused in a pattern match or function parameter list.

By convention, source files use two-character indentation for nesting—and they use spaces, not tabs, to achieve this.

Comments start with a hash sign (#) and

run to the end of the line.

The Elixir distribution comes with a code formatter, which can be used to convert a source file into the “approved” representation. We’ll look at this here. Most examples in this book follow this format (except where I think it is particularly ugly).

Truth

Elixir has three special values related to Boolean operations: true, false, and nil. nil is treated as false in Boolean contexts.

(A bit of trivia: all three of these values are aliases for atoms of the same name, so true is the same as the atom :true.)

In most contexts, any value other than false or nil is treated as true. We sometimes refer to this as *truthy* as opposed to true.

Operators

Elixir has a very rich set of operators. Here's a subset we'll use in this book:

Comparison operators

The ordering comparisons in Elixir are less strict than in many languages, as you can compare values of different types. If the types are the same or are compatible (for example, `3 > 2` or `3.0 < 5`), the comparison uses natural ordering. Otherwise comparison is

based on type according to this rule:

number < atom < reference < function < port < pid < tuple < map < list < binary

Boolean operators

(These operators expect true or false as their first argument.)

Relaxed Boolean operators

These operators take arguments of any type. Any value apart from nil or false is interpreted as true.

Arithmetic operators

+ - * / div rem

Integer division yields a floating-point result. Use div(a,b) to get an integer.

rem is the *remainder operator*. It is called as a function (rem(11, 3) => 2). It differs from normal modulo operations in that the result will have the same sign as the function's first argument.

Join operators

The in operator

Variable Scope

Elixir is lexically scoped. The basic unit of scoping is the function body. Variables defined in a function (including its parameters) are local to that function. In addition, modules define a scope for local variables, but these are accessible only at the top level of that module, and not in functions defined in the module.

Do-block Scope

Most languages let you group together multiple code statements and treat them as a single *code block*. Often languages use braces for this. Here's an example in C:

Elixir doesn't really have blocks such as these, but it does have ways of grouping expressions together. The most common of these is the do block:

However, Elixir thinks this is a risky way to write code. In particular, it's easy to forget to initialize line_no outside the block, but to then rely on it having a value after the block. For that reason, you'll see a warning:

The with Expression

The with expression serves double duty. First, it allows you to define a local scope for variables. If you need a couple

of temporary variables when calculating something, and you don't want those variables to leak out into the wider scope, use with. Second, it gives you some control over pattern-matching failures. For example, the /etc/passwd file contains lines such as

The two numbers are the user and group IDs for the given username.

The following code finds the values for the _lp user (and see the sidebar following for some notes on its layout).

basic-types/with-scope.exs

A Code Formatting Comparison

The with listing is an example of code that isn't in the canonical Elixir format. If formatted using the built-in tool, it would look like the following.

basic-types/with-scope-fmt.exs

I'll let you be the judge of which is clearer.

The with expression lets us work with what are effectively temporary variables as we open the file, read its content, close it, and search for the line we want. The value of the with is the value of its do parameter.

The inner variable content is local to the with, and does not affect the variable in the outer scope.

with and Pattern Matching

In the previous example, the head of the with expression used = for basic pattern

matches. If any of these had failed, a MatchError exception would be raised. But perhaps we'd want to handle this case in a more elegant way. That's where the <- operator comes in. If you use <- instead of = in a with expression, it performs a match, but if it fails it returns the value that couldn't be matched.

We can use this to let the with in the previous example return nil if the user can't be found, rather than raising an exception.

basic-types/with-match.exs

When we try to match the user xxx,

Regex.run returns nil. This causes the match to fail, and the nil becomes the value of the with.

A Minor Gotcha

Underneath the covers, with is treated by Elixir as if it were a call to a function or macro. This means that you cannot write this:

Instead, you can put the first parameter on the same line as the with:

or use parentheses:

As with all other uses of do, you can

also use the shortcut:

End of the Basics

We've now covered the low-level ingredients of an Elixir program. In the next two chapters we'll discuss how to create anonymous functions, modules, and named functions.

Footnotes

[4] http://www.pcre.org/

[5] http://www.iso.org/iso/home/standards/iso8601.htm

[6] https://github.com/lau/calendar

Copyright © 2018, The Pragmatic Bookshelf.

We cover:

- *Anonymous functions*
- *Pattern matching and arguments*
- *Higher-order functions*
- *Closures*
- *The & function literal*

Chapter 5

Anonymous Functions

Elixir is a functional language, so it's no surprise functions are a basic type.

An anonymous function is created using the fn keyword.

Think of fn…end as being a bit like the quotes that surround a string literal, except here we're returning a function as a value, not a string. We can pass that function value to other functions. We can also invoke it, passing in arguments.

At its simplest, a function has a parameter list and a body, separated by ->.

For example, the following defines a function, binding it to the variable sum, and then calls it:

The first line of code creates a function that takes two parameters (named a and

b). The implementation of the function follows the -> arrow (in our case it simply adds the two parameters), and the whole thing is terminated with the keyword end. We store the function in the variable sum.

On the second line of code, we invoke the function using the syntax sum.(1,2). The dot indicates the function call, and the arguments are passed between parentheses. (You'll have noticed we don't use a dot for named function calls —this is a difference between named and anonymous functions.)

If your function takes no arguments, you still need the parentheses to call it:

You can, however, omit the parentheses in a function definition:

2=

Functions and Pattern Matching

When we call sum.(2,3), it's easy to assume we simply assign 2 to the parameter a and 3 to b. But that word, *assign*, should ring some bells. Elixir doesn't have assignment. Instead it tries to match values to patterns. (We came across this when we looked at pattern matching and assignment.)

If we write

then Elixir makes the pattern match by binding a to the value 2. And that's exactly what happens when our sum function gets called. If we pass 2 and 3

as arguments, and Elixir tries to match these arguments to the parameters a and b (which it does by giving a the value 2 and b the value 3), it's the same as when we write

This means we can perform more complex pattern matching when we call a function. For example, the following function reverses the order of elements in a two-element tuple:

We'll use this pattern-matching capability when we look at functions with multiple implementations in the next section.

Your Turn

- Exercise: Functions-1

  Go into IEx. Create and run the functions that do the following:

  - `list_concat.([:a, :b], [:c, :d]) #=> [:a, :b, :c, :d]`
  - `sum.(1, 2, 3) #=> 6`
  - `pair_tuple_to_list.( { 1234, 5678 } ) #=> [ 1234, 5678 ]`

One Function, Multiple Bodies

A single function definition lets you define different implementations, depending on the type and contents of the arguments passed. (You cannot select based on the number of arguments—each clause in the function definition must have the same number of parameters.)

At its simplest, we can use pattern matching to select which clause to run. In the example that follows, we know the tuple returned by `File.open` has `:ok` as its first element if the file was opened, so we write a function that displays either the first line of a successfully

opened file or a simple error message if the file could not be opened.

Start by looking inside the function definition. On lines 2 and 3 we define two separate function bodies. Each takes a single tuple as a parameter. The first of them requires that the first term in the tuple is :ok. The second line uses the special variable _ (underscore) to match any other value for the first term.

Now look at line 6. We call our function, passing it the result of calling File.open on a file that exists. This means the function will receive the tuple {:ok,file}, and this matches the clause on line 2. The corresponding code calls

IO.read to read the first line of this file.

We then call handle_open again, this time with the result of trying to open a file that does not exist. The tuple that is returned ({:error,:enoent}) is passed to our function, which looks for a matching clause. It fails on line 2 because the first term is not :ok, but it succeeds on the next line. The code in that clause formats the error as a nice string.

Note a couple of other things in this code. On line 3 we call :file.format_error. The :file part of this refers to the underlying Erlang File module, so we can call its format_error function. Contrast this with the call to File.open on line 6. Here the File part

refers to Elixir's built-in module. This is a good example of the underlying environment leaking through into Elixir code. It is good that you can access all the existing Erlang libraries—there are hundreds of years of effort in there just waiting for you to use. But it is also tricky because you have to differentiate between Erlang functions and Elixir functions when you call them.

And finally, this example shows off Elixir's *string interpolation*. Inside a string, the contents of `#{...}` are evaluated and the result is substituted back in.

Working with Larger Code Examples

Our `handle_open` function is getting uncomfortably long to type directly into IEx. One typo, and we'd have to type it all in again.

Instead, let's use our editor to type it into a file in the same directory we were in when we started IEx. Let's call the file handle_open.exs.

first_steps/handle_open.exs

Now, inside IEx, type this:

This compiles and runs the code in the given file.

We can do the same thing from the command line (that is, not inside IEx) using this:

We used the file extension .exs for this example. This is used for code that we want to run directly from a source file (think of the *s* as meaning *script*). For files we want to compile and use later, we'll employ the .ex extension.

Your Turn

- Exercise: Functions-2

  Write a function that takes three arguments. If the first two are zero, return “FizzBuzz.” If the first is zero, return “Fizz.” If the second is zero, return “Buzz.” Otherwise return the third argument. Do not use any language features that we haven’t yet covered in this book.

- Exercise: Functions-3

  The operator `rem(a, b)` returns the remainder after dividing `a` by `b`. Write a function that takes a single integer (`n`) and calls the function in the previous exercise, passing it `rem(n,3)`, `rem(n,5)`, and `n`. Call it seven times with the arguments 10, 11, 12, and so on. You should get “Buzz, 11, Fizz, 13, 14, FizzBuzz, 16.”

  (Yes, it’s a FizzBuzz solution with no conditional logic.)[7]

Functions Can Return Functions

Here's some strange code:

```
```

The strange thing is the first line. It's hard to read, so let's spread it out.

```
```

The variable fun1 is bound to a function. That function takes no parameters, and its body is a second function definition. That second function also takes no parameters, and it evaluates the string "Hello".

When we call the outer function (using fun1.()), it returns the inner function.

When we call that (fun1.().()) the inner function is evaluated and “Hello” is returned.

We wouldn’t normally write something such as fun1.().(). But we might call the outer function and bind the result to a separate variable. We might also use parentheses to make the inner function more obvious.

Functions Remember Their Original Environment

Let’s take this idea of nesting functions a little further.

Now the outer function has a name parameter. Like any parameter, name is

available for use throughout the body of the function. In this case, we use it inside the string in the inner function.

When we call the outer function, it returns the inner function definition. It has not yet substituted the name into the string. But when we call the inner function (`dave_greeter.()`), the substitution takes place and the greeting appears.

But something strange happens here. The inner function uses the outer function's `name` parameter. But by the time `greeter.("Dave")` returns, that outer function has finished executing and the parameter has gone out of scope. And yet when we run the inner function, it uses that parameter's value.

This works because functions in Elixir automatically carry with them the bindings of variables in the scope in which they are defined. In our example, the variable name is bound in the scope of the outer function. When the inner function is defined, it inherits this scope and carries the binding of name around with it. This is a *closure*—the scope encloses the bindings of its variables, packaging them into something that can be saved and used later.

Let's play with this some more.

Parameterized Functions

In the previous example, the outer function took an argument and the inner one did not. Let's try a different example where both take arguments.

Here the inner function adds the value of its parameter other to the value of the outer function's parameter n. Each time we call the outer function, we give it a value for n and it returns a function that adds n to its own parameter.

#Function<erl_eval.6.82930912>

Your Turn

- Exercise: Functions-4

  Write a function prefix that takes a string. It should return a new function that takes a second string. When that second function is called, it will return a string containing the first string, a space, and the second string.

Passing Functions as Arguments

Functions are just values, so we can pass them to other functions.

Here, apply is a function that takes a second function and a value. It returns the result of invoking that second function with the value as an argument.

We use the ability to pass functions around pretty much everywhere in Elixir code. For example, the built-in Enum module has a function called map. It takes two arguments: a collection and a function. It returns a list that is the result of applying that function to each

element of the collection.

Pinned Values and Function Parameters

When we originally looked at pattern matching, we saw that the pin operator (^) allowed us to use the current value of a variable in a pattern. You can use this with function parameters, too.

functions/pin.exs

Here, the Greeter.for function returns a function with two heads. The first head matches when its first parameter is the value of the name passed to for.

The & Notation

The strategy of creating short helper

functions is so common that Elixir provides a shortcut. Let's look at it in use before we explore what's going on.

The `&` operator converts the expression that follows into a function. Inside that expression, the placeholders `&1`, `&2`, and so on correspond to the first, second, and subsequent parameters of the function. So `&(&1 + &2)` will be converted to `fn p1, p2 -> p1 + p2 end`.

If you think that's clever, take a look at the `speak` line in the previous code. Normally Elixir would have generated an anonymous function, so `&(IO.puts(&1))` would become `fn x -> IO.puts(x) end`. But Elixir noticed that

the body of the anonymous function was simply a call to a named function (the IO function puts) and that the parameters were in the correct order (that is, the first parameter to the anonymous function was the first parameter to the named function, and so on). So Elixir optimized away the anonymous function, replacing it with a direct reference to the function, IO.puts/1.

For this to work, the arguments must be in the correct order:

You might see references to Erlang pop up when you define functions this way. That's because Elixir runs on the Erlang

VM. There's more evidence of this if you try something like &abs(&1). Here Elixir maps your use of the abs function directly into the underlying Erlang library, and returns &:erlang.abs/1.

Because [] and {} are operators in Elixir, literal lists and tuples can also be turned into functions. Here's a function that returns a tuple containing the quotient and remainder of dividing two integers:

Finally, the & capture operator works with string (and string-like) literals:

There's a second form of the & function

capture operator. You can give it the name and arity (number of parameters) of an existing function, and it will return an anonymous function that calls it. The arguments you pass to the anonymous function will in turn be passed to the named function. We've already seen this: when we entered `&(IO.puts(&1))` into iex, it displayed the result as `&IO.puts/1`. In this case, `puts` is a function in the `IO` module, and it takes one argument. The Elixir way of naming this is `IO.puts/1`. If we place an `&` in front of this, we wrap it in a function. Here are some other examples:

This works with named functions we write, as well (but we haven't covered

how to write them yet).

The & shortcut gives us a wonderful way to pass functions to other functions.

Your Turn

- Exercise: Functions-5

  Use the & notation to rewrite the following:

  - `Enum.map [1,2,3,4], fn x -> x + 2 end`
  - `Enum.each [1,2,3,4], fn x -> IO.inspect x end`

Functions Are the Core

At the start of the book, we said the basis of programming is transforming data. Functions are the little engines that perform that transformation. They are at the very heart of Elixir.

So far we've been looking at anonymous functions—although we can bind them to variables, the functions themselves have no names. Elixir also has named functions. In the next chapter we'll cover how to work with them.

Footnotes

[7] http://c2.com/cgi/wiki?FizzBuzzTest

Copyright © 2018, The Pragmatic Bookshelf.

We cover:

- *Modules, the basic units of code*
- *Defining public and private named functions*
- *Guard clauses*
- *Module directives and attributes*
- *Calling functions in Erlang modules*

Chapter 6

Modules and Named Functions

Once a program grows beyond a couple of lines, you'll want to structure it.

Elixir makes this easy. You break your code into *named functions* and organize these functions into *modules*. In fact, in Elixir named functions *must* be written inside modules.

Let's look at a simple example. Navigate to a working directory and create an Elixir source file called times.exs.

mm/times.exs

Here we have a module named Times. It contains a single function, double. Because our function takes a single argument and because the number of arguments forms part of the way we identify Elixir functions, you'll see this

function name written as double/1.

[illegible]uble(42)

Compiling a Module

Let’s look at two ways to compile this file and load it into IEx. First, if you’re at the command line, you can do this:

Give IEx a source file’s name, and it compiles and loads the file before it displays a prompt.

If you’re already in IEx, you can use the c helper to compile your file without returning to the command line.

The line c "times.exs" compiles your source file and loads it into IEx. We

then call the double function in the Times module a couple of times using Times.double.

What happens if we make our function fail by passing it a string rather than a number?

An exception (ArithmeticError) gets raised, and we see a stack backtrace. The first line tells us what went wrong (we tried to perform arithmetic on a string), and the next line tells us where. But look at what it writes for the name of our function: Times.double/1.

In Elixir a named function is identified by both its name and its number of

parameters (its *arity*). Our double function takes one parameter, so Elixir knows it as double/1. If we had another version of double that took three parameters, it would be known as double/3. These two functions are totally separate as far as Elixir is concerned. But from a human perspective, you'd imagine that if two functions have the same name they are somehow related, even if they have a different number of parameters. For that reason, don't use the same name for two functions that do unrelated things.

greeting,

The Function's Body Is a Block

The do…end block is one way of grouping expressions and passing them to other code. They are used in module and named function definitions, control structures…any place in Elixir where code needs to be handled as an entity.

However, do…end is not actually the underlying syntax. The actual syntax looks like this:

You can pass multiple lines to do: by grouping them with parentheses.

The do…end form is just a lump of syntactic sugar—during compilation it is turned into the do: form. (And the do: form itself is nothing special; it is simply a term in a keyword list.) Typically people use the do: syntax for single-line blocks, and do…end for multiline ones.

This means our times example would probably be written as follows:

mm/times1.exs

We could even write it as

(but please don't).

Your Turn

- Exercise: ModulesAndFunctions-1

  Extend the Times module with a triple function that multiplies its parameter by three.

- Exercise: ModulesAndFunctions-2

  Run the result in IEx. Use both techniques to compile the file.

- Exercise: ModulesAndFunctions-3

  Add a quadruple function. (Maybe it could call the double function.…)

Function Calls and Pattern Matching

In the previous chapter we covered how anonymous functions use pattern matching to bind their parameter list to the passed arguments. The same is true of named functions. The difference is that we write the function multiple times, each time with its own parameter list and body. Although this looks like multiple function definitions, purists will tell you it's multiple clauses of the same definition (and they'd be right).

When you call a named function, Elixir tries to match your arguments with the parameter list of the first definition (clause). If it cannot match them, it tries

the next definition of the same function (remember, this must have the same arity) and checks to see if it matches. It continues until it runs out of candidates.

Let's play with this. The factorial of *n*, written *n!*, is the product of all numbers from 1 to *n*. 0! is defined to be 1.

Another way of expressing this is to say

- factorial(0) → 1
- factorial(n) → n * factorial(n-1)

This is a specification of the concept of *factorial*, but it is also close to an Elixir implementation:

mm/factorial1.exs

Here we have two definitions *of the same function*. If we call Factorial.of(2), Elixir matches the 2 against the first function's parameter, 0. This fails, so it tries the second definition, which succeeds when Elixir binds 2 to n. It then evaluates the body of this function, which calls Factorial.of(1). The same process applies, and the second definition is run. This, in turn, calls Factorial.of(0), which is matched by the first function definition. This function returns 1 and the recursion ends. Elixir now unwinds the stack, performing all the multiplications, and returns the answer. This factorial implementation works, but it could be significantly improved. We'll do that improvement when we look at tail recursion.

Let’s play with this code:

This pattern of design and coding is very common in Elixir (and almost all functional languages). First look for the simplest possible case, one that has a definite answer. This will be the anchor. Then look for a recursive solution that will end up calling the anchor case.

Here are a couple of examples:

Sum of the first n numbers

- The sum of the first 0 numbers is 0.
- The sum of the numbers up to n is n + the sum of the numbers up to $n–1$.

Length of a list

- The length of an empty list is 0.
- The length of any other list is 1 + the length of the tail of that list.

One point worth stressing: the order of these clauses can make a difference when you translate them into code. Elixir tries functions from the top down, executing the first match. So the following code will not work:

mm/factorial1-bad.exs

The first function definition will always match and the second will never be called. But Elixir has you covered—when you try to compile this, you'll get a warning:

One more thing: when you have multiple implementations of the same function, they should be adjacent in the source file.

Your Turn

- Exercise: ModulesAndFunctions-4

  Implement and run a function sum(n) that uses recursion to calculate the sum of the integers from 1 to *n*. You'll need to write this function inside a module in a separate file. Then load up IEx, compile that file, and try your function.

- Exercise: ModulesAndFunctions-5

  Write a function gcd(x,y) that finds the greatest common divisor between two nonnegative integers. Algebraically, *gcd(x,y)* is *x* if *y* is zero; it's *gcd(y, rem(x,y))* otherwise.

[1,2,3])

Guard Clauses

We've seen that pattern matching allows Elixir to decide which function to invoke based on the arguments passed. But what if we need to distinguish based on the argument types or on some test involving their values? For this, use *guard clauses*. These are predicates that are attached to a function definition using one or more when keywords. When doing pattern matching, Elixir first does the conventional parameter-based match and then evaluates any when predicates, executing the function only if at least one predicate is true.

mm/guard.exs

Recall our factorial example:

mm/factorial1.exs

If we were to pass it a negative number, it would loop forever—no matter how many times you decrement n, it will never be zero. So it is a good idea to add a guard clause to stop this from happening.

mm/factorial2.exs

If you run this with a negative argument, none of the functions will match:

Notice we've also added a type constraint: the parameter must be an

integer.

Guard Clauses vs. Conditional Logic

Have another look at our factorial function:

You might think of writing the code like this:

Logically these are the same, right? Both versions raise an exception when passed a negative number.

But they aren't. In the second case, the `of/1` function is defined for any input. But in the first case, it isn't defined at all for negative parameters. And that's what we really want: the first example makes it explicit that the domain of our function is nonnegative integers.

The difference between the two examples is subtle, but the first communicates what we want more accurately.

Guard-Clause Limitations

You can write only a subset of Elixir expressions in guard clauses. The following list comes from the Getting Started guide.[8]

Comparison operators

==, !=, ===, !==, >, <, <=, >=

Boolean and negation operators

or, and, not, !. Note that || and && are not allowed.

Arithmetic operators

+, -, *, /

Join operators

<> and ++, as long as the left side is a literal

The in operator

Membership in a collection or range

Type-check functions

These built-in Erlang functions

return true if their argument is a given type. You can find their documentation online.[9]

Other functions

These built-in functions return values (not true or false). Their documentation is online, on the same page as the type-check functions.

Default Parameters

When you define a named function, you can give a default value to any of its parameters by using the syntax param \\ value. When you call a function that is defined with default parameters, Elixir compares the number of arguments you are passing with the number of required parameters for the function. If you're passing fewer arguments than the number of required parameters, then there's no match. If the two numbers are equal, then the required parameters take the values of the passed arguments, and the other parameters take their default values. If the count of passed arguments is greater than the number of required

parameters, Elixir uses the excess to override the default values of some or all parameters. Parameters are matched left to right.

mm/default_params.exs

Default arguments can behave surprisingly when Elixir does pattern matching. For example, compile the following:

and you'll get this error:

That's because the first function definition (with the default parameters) matches any call with two, three, or four

arguments.

There's one more thing with default parameters. Here's a function with multiple heads that also has a default parameter:

mm/default_params1.exs

If you compile this, you'll get an error:

The intent is to reduce confusion that can arise with defaults. Add a function head with no body that contains the default parameters, and use regular parameters for the rest. The defaults will apply to all calls to the function.

mm/default_params2.exs

guess(273,

Your Turn

- Exercise: ModulesAndFunctions-6

  I'm thinking of a number between 1 and 1000.
  …

  The most efficient way to find the number is to guess halfway between the low and high numbers of the range. If our guess is too big, then the answer lies between the bottom of the range and one less than our guess. If our guess is too small, then the answer lies between one more than our guess and the end of the range.

  Your API will be guess(actual, range), where range is an Elixir range. Your output should look similar to this:

  Hints:

  - You may need to implement helper functions with an additional parameter (the currently guessed number).

- The `div(a,b)` function performs integer division.
- Guard clauses are your friends.
- Patterns can match the low and high parts of a range (`a..b = 4..8`).

fa(a),

Private Functions

The defp macro defines a private function—one that can be called only within the module that declares it.

You can define private functions with multiple heads, just as you can with def. However, you cannot have some heads private and others public. That is, the following code is not valid:

```
[illegible] = Orders.for_customers(DB.find_customers),
```

The Amazing Pipe Operator: |>

I've saved the best for last, at least when it comes to functions.

We've all seen code like this:

Bread-and-butter programming. We did it because the alternative was to write

and that's the kind of code that you use to get kids to eat their vegetables. Not only is it hard to read, but you have to read it inside out if you want to see the order in which things get done.

Elixir has a better way of writing it:

The `|>` operator takes the result of the expression to its left and inserts it as the first parameter of the function invocation to its right. So the list of customers the first call returns becomes the argument passed to the `for_customers` function. The resulting list of orders becomes the first argument to `sales_tax`, and the given parameter, 2018, becomes the second.

`val |> f(a,b)` is basically the same as calling `f(val,a,b)`, and

is the same as

prepare_filing(sales_tax(list, 2018)).

In the previous example, I wrote each term in the expression on a separate line, and that's perfectly valid Elixir. But you can also chain terms on the same line:

Note that I had to use parentheses in that code—the & shortcut and the pipe operator fight otherwise.

Let me repeat that—you should always use parentheses around function parameters in pipelines.

The key aspect of the pipe operator is that it lets you write code that pretty much follows your spec's form. For the sales-tax example, you might have

jotted this on some paper:

- Get the customer list.
- Generate a list of their orders.
- Calculate tax on the orders.
- Prepare the filing.

To take this from a napkin spec to running code, you just put `|>` between the items and implement each as a function.

Programming is transforming data, and the `|>` operator makes that transformation explicit.

And now this book's subtitle makes sense.

Modules

Modules provide namespaces for things you define. We've already seen them encapsulating named functions. They also act as wrappers for macros, structs, protocols, and other modules.

If we want to reference a function defined in a module from outside that module, we need to prefix the reference with the module's name. We don't need that prefix if code references something inside the same module as itself, as in the following example:

func2 can call func1 directly because it

is inside the same module. Outside the module, you have to use the fully qualified name, Mod.func1.

Just as you do in your favorite language, Elixir programmers use nested modules to impose structure for readability and reuse. After all, every programmer is a library writer.

To access a function in a nested module from the outside scope, prefix it with all the module names. To access it within the containing module, use either the fully qualified name or just the inner module name as a prefix.

Module nesting in Elixir is an illusion—

all modules are defined at the top level. When we define a module inside another, Elixir simply prepends the outer module name to the inner module name, putting a dot between the two. This means we can directly define a nested module.

It also means there's no particular relationship between the modules Mix and Mix.Tasks.Doctest.

Directives for Modules

Elixir has three directives that simplify working with modules. All three are executed as your program runs, and the effect of all three is *lexically scoped*—it starts at the point the directive is

encountered, and stops at the end of the enclosing scope. This means a directive in a module definition takes effect from the place you wrote it until the end of the module; a directive in a function definition runs to the end of the function.

The import Directive

The import directive brings a module's functions and/or macros into the current scope. If you use a particular module a lot in your code, import can cut down the clutter in your source files by eliminating the need to repeat the module name time and again.

For example, if you import the flatten function from the List module, you'd be able to call it in your code without

having to specify the module name.

mm/import.exs

The full syntax of import is

The optional second parameter lets you control which functions or macros are imported. You write only: or except:, followed by a list of name: arity pairs. It is a good idea to use import in the smallest possible enclosing scope and to use only: to import just the functions you need.

Alternatively, you can give only: one of the atoms :functions or :macros, and

import will bring in only functions or macros.

The alias Directive

The alias directive creates an alias for a module. One obvious use is to cut down on typing.

We could have abbreviated these alias directives to

because the as: parameters default to the last part of the module name. We could even take this further, and do this:

The require Directive

You require a module if you want to use any macros it defines. This ensures that the macro definitions are available when your code is compiled. We'll talk about require when we discuss macros.

Module Attributes

Elixir modules each have associated metadata. Each item of metadata is called an *attribute* of the module and is identified by a name. Inside a module, you can access these attributes by prefixing the name with an at sign (@). You give an attribute a value using the syntax

This works only at the top level of a module—you can’t set an attribute inside a function definition. You can, however, access attributes inside functions.

mm/attributes.exs

You can set the same attribute multiple times in a module. If you access that attribute in a named function in that module, the value you see will be the value in effect when the function is defined.

mm/attributes1.exs

These attributes are not variables in the conventional sense. Use them for configuration and metadata only. (Many Elixir programmers employ them where Java or Ruby programmers might use constants.)

Module Names: Elixir, Erlang, and Atoms

When we write modules in Elixir, they have names such as `String` or `PhotoAlbum`. We call functions in them using calls such as `String.length("abc")`.

What's happening here is subtle. Internally, module names are just atoms. When you write a name starting with an uppercase letter, such as `IO`, Elixir converts it internally into an atom of the same name with `Elixir.` prepended. So `IO` becomes `Elixir.IO` and `Dog` becomes `Elixir.Dog`.

So a call to a function in a module is

really an atom followed by a dot followed by the function name. And, indeed, we can call functions like this:

and even

Calling a Function in an Erlang Library

The Erlang conventions for names are different—variables start with an uppercase letter and atoms are simple lowercase names. So, for example, the Erlang module timer is called just that, the atom timer. In Elixir we write that as :timer. If you want to refer to the tc function in timer, you'd write :timer.tc. (Note the colon at the start.)

Say we want to output a floating-point number in a three-character-wide field with one decimal place. Erlang has a function for this. A search for erlang format takes us to the description of the format function in the Erlang io module. [10]

Reading the description, we see that Erlang expects us to call io.format. So, in Elixir we simply change the Erlang module name to an Elixir atom:

Finding Libraries

If you're looking for a library to use in your app, you'll want to look first for existing Elixir modules. The built-in ones are documented on the Elixir website,[11] and others are listed at http://hex.pm and on GitHub (search for *elixir*).

If that fails, search for a built-in Erlang library or search the web.[12] If you find something written in Erlang, you'll be able to use it in your project (we'll cover how in the chapter on projects,). But be aware that the Erlang documentation for a library follows Erlang conventions. Variables start with uppercase letters,

and identifiers that start with a lowercase letter are atoms (so Erlang would say tomato and Elixir would say :tomato). A summary of the differences between Elixir and Erlang is available online.[13]

Now that we've looked at functions, let's move on to the data they manipulate. And where better to start than with lists? They're the subject of the next chapter.

Your Turn

- Exercise: ModulesAndFunctions-7

  Find the library functions to do the following, and then use each in IEx. (If the word *Elixir* or *Erlang* appears at the end of the challenge, then you'll find the answer in that set of libraries.)

  - Convert a float to a string with two decimal digits. (Erlang)
  - Get the value of an operating-system environment variable. (Elixir)
  - Return the extension component of a file name (so return `.exs` if given `"dave/test.exs"`). (Elixir)
  - Return the process's current working directory. (Elixir)
  - Convert a string containing JSON into Elixir data structures. (Just find; don't install.)
  - Execute a command in your

operating system's shell.

Footnotes

[8] http://elixir-lang.org/getting-started/case-cond-and-if.html#expressions-in-guard-clauses

[9] http://erlang.org/doc/man/erlang.html#is_atom-1

[10] http://erlang.org/doc/man/io.html#format-2

[11] http://elixir-lang.org/docs/

[12] http://erlang.org/doc/ and http://erldocs.com/19.0/ (Note that the latter is slightly out of date.)

[13] http://elixir-lang.org/crash-course.html

Copyright © 2018, The Pragmatic Bookshelf.

We cover:

- *The recursive structure of lists*
- *Traversing and building lists*
- *Accumulators*
- *Implementing map and reduce*

Chapter 7

Lists and Recursion

When we program with lists in conventional languages, we treat them as things to be iterated—it seems natural to loop over them. So why do we have a chapter on *lists and recursion*? Because if you look at the problem in the right way, recursion is a perfect tool for

processing lists.

Heads and Tails

Earlier we said a list may either be empty or consist of a head and a tail. The head contains a value and the tail is itself a list. This is a recursive definition.

We'll represent the empty list like this: `[ ]`.

Let's imagine we could represent the split between the head and the tail using a pipe character: `|`. The single-element list we normally write as `[ 3 ]` can be written as the value `3` joined to the empty list:

(I've highlighted the inner list.)

When we see the pipe character, we say that what's on the left is the head of a list and what's on the right is the tail.

Let's look at the list [2, 3]. The head is 2, and the tail is the single-element list containing 3. And we know what that list looks like—it is our previous example. So we could write [2,3] as

At this point, part of your brain is telling you to go read today's XKCD—this list stuff can't be useful. Ignore that small voice, just for a second. We're about to do something magical. But before we do, let's add one more term, making our

list [1, 2, 3]. This is the head 1 followed by the list [2, 3], which is what we derived a moment ago:

This is valid Elixir syntax. Type it into IEx.

And here's the magic. When we discussed pattern matching, we said the pattern could be a list, and the values in that list would be assigned from the right-hand side.

We can also use the pipe character in the pattern. What's to the left of it matches the head value of the list, and what's to

the right matches the tail.

How IEx Displays Lists

In Chapter 11, *Strings and Binaries*, you'll see that Elixir has two representations for strings. One is the familiar sequence of characters in consecutive memory locations. Literals written with double quotes use this form.

The second form, using single quotes, represents a string as a list of integer codepoints. So the string `'cat'` is the three codepoints: 99, 97, and 116.

This is a headache for IEx. When it sees a list like `[99,97,116]` it doesn't know if it is supposed to be the string `'cat'` or a list of three numbers. So it uses a heuristic. If all the values in a list represent printable characters, it displays the list as a string; otherwise it displays a list of integers.

In Chapter 11, *Strings and Binaries*, we'll cover how to bypass this behavior. In the meantime, don't be surprised if a string pops up when you were expecting a list.

Using Head and Tail to Process a List

Now we can split a list into its head and its tail, and we can construct a list from a value and a list, which become the head and tail of that new list.

So why talk about lists after we talk about modules and functions? Because lists and recursive functions go together like fish and chips. Let's look at finding the length of a list.

- The length of an empty list is 0.
- The length of a list is 1 plus the length of that list's tail.

Writing the list-length algorithm in Elixir is easy:

lists/mylist.exs

The only tricky part is the definition of the function's second variant:

This is a pattern match for any nonempty list. When it does match, the variable head will hold the value of the first element of the list, and tail will hold the rest of the list. (And remember that every list is terminated by an empty list, so the tail can be [].)

Let's see this at work with the list [11, 12, 13, 14, 15]. At each step, we take off the head and add 1 to the length of the tail:

Let's try our code to see if theory works in practice:

It works, but we have a compilation warning—we never used the variable head in the body of our function. We can fix that, and make our code more explicit, using the special variable _ (underscore), which acts as a placeholder. We can also use an underscore in front of any variable name to turn off the warning if that variable isn't used. I sometimes like to do this to document the unused parameter.

lists/mylist1.exs

When we compile, the warning is gone.

(However, if you compile the second version of MyList, you may get a warning about “redefining module MyList.” This is just Elixir being cautious.)

Using Head and Tail to Build a List

Let's get more ambitious. Let's write a function that takes a list of numbers and returns a new list containing the square of each. We don't show it, but these definitions are also inside the MyList module.

lists/mylist1.exs

There's a lot going on here. First, look at the parameter patterns for the two definitions of square. The first matches an empty list and the second matches all other lists.

Second, look at the body of the second

definition:

When we match a nonempty list, we return a new list whose head is the square of the original list's head and whose tail is a list of squares of the tail. This is the recursive step.

Let's try it:

Let's do something similar—a function that adds 1 to every element in the list:

lists/mylist1.exs

And call it:

Creating a Map Function

With both square and add_1, all the work is done in the second function definition. And that definition looks about the same for each—it returns a new list whose head is the result of either squaring or incrementing the head of its argument and whose tail is the result of calling itself recursively on the tail of the argument. Let's generalize this. We'll define a function called map that takes a list and a function and returns a new list containing the result of applying that function to each element in the original.

lists/mylist1.exs

The map function is pretty much identical to the square and add_1 functions. It returns an empty list if passed an empty list; otherwise it returns a list where the head is the result of calling the passed-in function and the tail is a recursive call to itself. Note that in the case of an empty list, we use _func as the second parameter. The underscore prevents Elixir from warning us about an unused variable.

To call this function, pass in a list and a function (defined using fn):

A function is just a built-in type, defined between fn and the end. Here we pass a function as the second argument (func)

to map. This is invoked inside map using func.(head), which squares the value in head, using the result to build the new list.

We can call map with a different function:

And another:

And we can do the same using the & shortcut notation:

Reducing a List to a Single Value

The map/2 function we just wrote abstracts the idea of applying a function to each element of a list independently.

But what if we want to apply that function across the elements? How could we create an abstraction that would let us sum a list, or find the product of its elements, or find the largest element?

The sum function reduces a collection to a single value. Other functions need to do something similar—return the greatest/least value, the product of the elements, a string containing a

concatenation of elements, and so on. How can we write a general-purpose function that reduces a collection to a value?

We know it has to take a collection. We also know we need to pass in some initial value (just like our sum/1 function passed a 0 as an initial value to its helper). Additionally, we need to pass in a function that takes the current value of the reduction along with the next element of the collection, and returns the next value of the reduction. So, it looks like our reduce function will be called with three arguments:

Now let's think about the recursive

design:

- reduce([ ], value, _fun) → value
- reduce([head | tail], value, fun) → reduce(tail, fun.(head, value), fun)

reduce applies the function to the list's head and the current value, and passes the result as the new current value when reducing the list's tail.

Here's our code for reduce. See how closely it follows the design:

lists/reduce.exs

And, again, we can use the shorthand notation to pass in the function:

Your Turn

- Exercise: ListsAndRecursion-1

  Write a mapsum function that takes a list and a function. It applies the function to each element of the list and then sums the result, so

- Exercise: ListsAndRecursion-2

  Write a max(list) that returns the element with the maximum value in the list. (This is slightly trickier than it sounds.)

- Exercise: ListsAndRecursion-3

  An Elixir single-quoted string is actually a list of individual character codes. Write a caesar(list, n) function that adds n to each list element, wrapping if the addition results in a character greater than *z*.

More Complex List Patterns

Not every list problem can be easily solved by processing one element at a time. Fortunately, the join operator, `|`, supports multiple values to its left. Thus, you could write

The same thing works in patterns, so you can match multiple individual elements as the head. For example, this program swaps pairs of values in a list:

lists/swap.exs

We can play with it in IEx:

The third definition of swap matches a list with a single element. This will happen if we get to the end of the recursion and have only one element left. Given that we take two values off the list on each cycle, the initial list must have had an odd number of elements.

Lists of Lists

Let's imagine we had recorded temperatures and rainfall at a number of weather stations. Each reading looks like this:

Our code is passed a list containing a number of these readings, and we want to report on the conditions for one

particular location, number 27.

lists/weather.exs

This is a standard *recurse until the list is empty* stanza. But look at our function definition's second clause. Where we'd normally match into a variable called head, here the pattern is

For this to match, the head of the list must itself be a four-element list, and the second element of this sublist must be 27. This function will execute only for entries from the desired location. But when we do this kind of filtering, we also have to remember to deal with the case when our function doesn't match.

That's what the third line does. We could have written

but in reality we don't care *what* is in the head at this point.

In the same module we define some simple test data:

lists/weather.exs

We can use that to play with our function in IEx. To make this easier, I'm using the import function. This adds the functions in WeatherHistory to our local name scope. After calling import we don't have to put the module name in front of every function call.

Our function is specific to a particular location, which is pretty limiting. We'd like to be able to pass in the location as a parameter. We can use pattern matching for this.

lists/weather2.exs

Now the second function fires only when the location extracted from the list head equals the target location passed as a parameter.

But we can improve on this. Our filter doesn't care about the other three fields in the head—it just needs the location. But we do need the value of the head itself to create the output list.

Fortunately, Elixir pattern matching is recursive and we can match patterns inside patterns.

lists/weather3.exs

The key change here is this line:

Compare that with the previous version:

In the new version, we use placeholders for the fields we don't care about. But we also match the entire four-element array into the parameter head. It's as if we said, "Match the head of the list where the second element is matched to target_loc and then match that whole

head with the variable head.” We’ve extracted an individual component of the sublist as well as the entire sublist.

In the original body of for_location, we generated our result list using the individual fields:

In the new version, we can just use the head, making it a lot clearer:

Your Turn

- Exercise: ListsAndRecursion-4

  Write a function `MyList.span(from, to)` that returns a list of the numbers from `from` up to `to`.

The List Module in Action

The List module provides a set of functions that operate on lists.

Get Friendly with Lists

Lists are the natural data structure to use when you have a stream of values to handle. You'll use them to parse data, handle collections of values, and record the results of a series of function calls. It's worth spending a while getting comfortable with them.

Next we'll look at the various dictionary types, including maps. These let us organize data into collections of key/value pairs.

Copyright © 2018, The Pragmatic Bookshelf.

We cover:

- *The two and a half dictionary data types*
- *Pattern matching and updating maps*
- *Structs*
- *Nested data structures*

Chapter 8

Maps, Keyword Lists, Sets, and Structs

A dictionary is a data type that associates keys with values.

We've already looked briefly at the dictionary types: maps and keyword

lists. In this short chapter we'll cover how to use them with pattern matching and how to update them. Then we'll dive into structs, a special kind of map with a fixed structure. Finally we'll explore nested data structures and see how to alter fields in a map inside another map inside another map.…

First, though, let's answer a common question—how do we choose an appropriate dictionary type for a particular need?

How to Choose Between Maps, Structs, and Keyword Lists

Ask yourself these questions (in this order):

- Do I want to pattern-match against the contents (for example, matching a dictionary that has a key of :name somewhere in it)?

  If so, use a map.

- Will I want more than one entry with the same key?

  If so, you'll have to use the Keyword module.

- Do I need to guarantee the elements are ordered?

  If so, again, use the Keyword module.

- Do I have a fixed set of fields (that is, is the structure of the data always the same)?

  If so, use a struct.

- Otherwise, if you've reached this point,

Use a map.

Keyword Lists

Keyword lists are typically used in the context of options passed to functions.

maps/keywords.exs

For simple access, you can use the access operator, *kwlist*[*key*]. In addition, all the functions of the Keyword and Enum modules are available.[14][15]

[illegible]

Maps

Maps are the go-to key/value data structure in Elixir. They have good performance at all sizes.

Let’s play with the Map API:[16]

ght:

Pattern Matching and Updating Maps

The question we most often ask of our maps is, “Do you have the following keys (and maybe values)?” For example, given this map:

- Is there an entry with the key :name?
- Are there entries for the keys :name and :height?
- Does the entry with key :name have the value "Dave"?

Our map does not have the key :weight,

so the following pattern match fails:

It's worth noting how the first pattern match destructured the map, extracting the value associated with the key :name. We can use this in many ways. Here's one example. The for construct lets us iterate over a collection, filtering as we go. We cover it when we talk about enumerating. The following example uses for to iterate over a list of people. Destructuring is used to extract the height value, which is used to filter the results.

maps/query.exs

This produces

In this code, we feed a list of maps to our comprehension. The generator clause binds each map (as a whole) to person and binds the height from that map to height. The filter selects only those maps where the height exceeds 1.5, and the do block returns the people that match. The comprehension as a whole returns a list of these people, which IO.inspect prints.

Clearly pattern matching is just pattern matching, so this maps capability works equally well in cond expressions, function head matching, and any other circumstances in which patterns are used.

maps/book_room.exs

Pattern Matching Can’t Bind Keys

You can’t bind a value to a key during pattern matching. You can write this:

but not this:

Pattern Matching Can Match Variable Keys

When we looked at basic pattern matching, we saw that the pin operator uses the value already in a variable on the left-hand side of a match. We can do the same with the keys of a map:

Updating a Map

Maps let us add new key/value entries and update existing entries without traversing the whole structure. But as with all values in Elixir, a map is immutable, and so the result of the update is a new map.

The simplest way to update a map is with this syntax:

This creates a new map that is a copy of the old, but the values associated with the keys on the right of the pipe character are updated:

However, this syntax will not add a new key to a map. To do this, you have to use the Map.put_new/3 function.

Structs

When Elixir sees `%{ … }` it knows it is looking at a map. But it doesn't know much more than that. In particular, it doesn't know what you intend to do with the map, whether only certain keys are allowed, or whether some keys should have default values.

That's fine for anonymous maps. But what if we want to create a typed map—a map that has a fixed set of fields and default values for those fields, and that you can pattern-match by type as well as content.

Enter the *struct*.

A struct is just a module that wraps a limited form of map. It's limited because the keys must be atoms and because these maps don't have Dict capabilities. The name of the module becomes the name of the map type.

Inside the module, you use the defstruct macro to define the struct's members.

maps/defstruct.exs

Let's play with this in IEx:

The syntax for creating a struct is the same as the syntax for creating a map—you simply add the module name between the % and the {.

You access the fields in a struct using dot notation or pattern matching:

And updates follow suit:

Why are structs wrapped in a module? The idea is that you are likely to want to add struct-specific behavior.

maps/defstruct1.exs

Notice how we could call the functions in the Attendee module to manipulate the associated struct.

Structs also play a large role when

implementing polymorphism, which we'll see when we look at protocols.

[illegible]:

Nested Dictionary Structures

The various dictionary types let us associate keys with values. But those values can themselves be dictionaries. For example, we may have a bug-reporting system. We could represent this using the following:

maps/nested.exs

Let's create a simple report:

The owner attribute of the report is itself a Customer struct.

We can access nested fields using

regular dot notation:

But now our customer complains the company name is incorrect—it should be PragProg. Let's fix it:

Ugly stuff! We had to update the overall bug report's owner attribute with an updated customer structure. This is verbose, hard to read, and error prone.

Fortunately, Elixir has a set of nested dictionary-access functions. One of these, put_in, lets us set a value in a nested structure:

This isn't magic—it's simply a macro that generates the long-winded code we'd have to have written otherwise.

The update_in function lets us apply a function to a value in a structure.

The other two nested access functions are get_in and get_and_update_in. The documentation in IEx contains everything you need for these. However, both of these functions support a cool trick: nested access.

Nested Accessors and Nonstructs

If you are using the nested accessor functions with maps or keyword lists, you can supply the keys as atoms:

Dynamic (Runtime) Nested Accessors

The nested accessors we've seen so far are macros—they operate at compile time. As a result, they have some limitations:

- The number of keys you pass a particular call is static.
- You can't pass the set of keys as parameters between functions.

These are a natural consequence of the way the macros bake their parameters into code at compile time.

To overcome this, get_in, put_in, update_in, and get_and_update_in can all take a list of keys as a separate parameter. Adding this parameter changes them from macros to function calls, so they become dynamic.

Here's a simple example:

maps/dynamic_nested.exs

There's a cool trick that the dynamic versions of both get_in and get_and_update_in support—if you pass a function as a key, that function is invoked to return the corresponding values.

maps/get_in_func.exs

The Access Module

The Access module provides a number of predefined functions to use as parameters to get and get_and_update_in.

These functions act as simple filters while traversing the structures.

The all and at functions only work on lists. all returns all elements in the list, while at returns the nth element (counting from zero).

maps/access1.exs

The elem function works on tuples:

maps/access2.exs

The key and key! functions work on dictionary types (maps and structs):

maps/access3.exs

Finally, Access.pop lets you remove the entry with a given key from a map or keyword list. It returns a tuple containing the value associated with the key and the updated container. nil is returned for the value if the key isn't in the container.

The name has nothing to do with the pop stack operation.

new)

Sets

Sets are implemented using the module MapSet.

With Great Power Comes Great Temptation

The dictionary types are clearly a powerful tool—you'll use them all the time. But you might also be tempted to abuse them. Structs in particular might lead you into the darkness because you can associate functions with them in their module definitions. At some point, the old object-orientation neurons still active in the nether regions of your brain might burst into life and you might think, "Hey, this is a bit like a class definition." And you'd be right. You *can* write something akin to object-oriented code using structs (or maps) and modules.

This is a bad idea—not because objects are intrinsically bad, but because you'll be mixing paradigms and diluting the benefits a functional approach gives you.

Stay pure, young coder. Stay pure.

As a way of refocusing you away from the dark side, the next chapter is a mini diversion into the benefits of separating functions and the data they work on. And we disguise it in a discussion of types.

Footnotes

[14] http://elixir-lang.org/docs/master/elixir/Keyword.html

[15] http://elixir-lang.org/docs/master/elixir/Enum.html

[16] http://elixir-lang.org/docs/master/elixir/Map.html

Copyright © 2018, The Pragmatic Bookshelf.

Chapter 9

An Aside—What Are Types?

The preceding two chapters described the basics of lists and maps. But you may have noticed that, although I talked about them as types, I didn't really say what I meant.

The first thing to understand is that the primitive data types are not necessarily the same as the types they can represent. For example, a primitive Elixir list is just an ordered group of values. We can use the `[…]` literal to create a list, and the `|` operator to deconstruct and build

lists.

Then there's another layer. Elixir has the List module, which provides a set of functions that operate on lists. Often these functions simply use recursion and the | operator to add this extra functionality.

In my mind, there's a difference between the primitive list and the functionality of the List module. The primitive list is an implementation, whereas the List module adds a layer of abstraction. Both implement types, but the type is different. Primitive lists, for example, don't have a flatten function.

Maps are also a primitive type. And, like lists, they have an Elixir module

that implements a richer, derived map type.

The code that provides the Keyword type is an Elixir module. But the type is represented as a list of tuples:

Clearly this is still a list, and all the list functions will work on it. But Elixir adds functionality to give you dictionary-like behavior.

In a way, this is a form of the duck typing that is talked about in dynamic object-oriented languages.[17] The Keyword module doesn't have an underlying primitive data type. It simply

assumes that any value it works on is a list that has been structured a certain way.

This means the APIs for collections in Elixir are fairly broad. Working with a keyword list, you have access to the APIs in the primitive list type, and the List and Keyword modules. You also get Enum and Collectable, which we talk about next.

Footnotes

[17] http://en.wikipedia.org/wiki/Duck_typing

Copyright © 2018, The Pragmatic Bookshelf.

We cover:

- *The Enum module*
- *The Stream module*
- *The Collectable protocol*
- *Comprehensions*

Chapter 10

Processing Collections —Enum and Stream

Elixir comes with a number of types that act as collections. We've already seen lists and maps. Ranges, files, and even functions can also act as collections. And as we'll discuss when we look at protocols, you can also define your own.

Collections differ in their implementation. But they all share something: you can iterate through them. Some of them share an additional trait: you can add things to them.

Technically, things that can be iterated are said to implement the Enumerable protocol.

Elixir provides two modules that have a bunch of iteration functions. The Enum module is the workhorse for collections. You'll use it all the time. I strongly recommend getting to know it.

The Stream module lets you enumerate a collection lazily. This means that the next value is calculated only when it is needed. You'll use this less often, but

when you do it's a lifesaver.

I don't want to fill this book with a list of all the APIs. You'll find the definitive (and up-to-date) list online.[18] Instead, I'll illustrate some common uses and let you browse the documentation for yourself. (But please do remember to do so. Much of Elixir's power comes from these libraries.)

[illegible],

Enum—Processing Collections

The Enum module is probably the most used of all the Elixir libraries. Employ it to iterate, filter, combine, split, and otherwise manipulate collections. Here are some common tasks:

- Convert any collection into a list:

- Concatenate collections:

- Create collections whose elements are some function of the original:

- Select elements by position or criteria:

- Sort and compare elements:

- Split a collection:

- Join a collection:

- Predicate operations:

- Merge collections:

- Fold elements into a single value:

- Deal a hand of cards:

A Note on Sorting

In our example of sort, we used

```
```

It's important to use `<=` and not just `<` if you want the sort to be *stable*.

```
[illegible]n([
```

Your Turn

- Exercise: ListsAndRecursion-5

  Implement the following Enum functions using no library functions or list comprehensions: all?, each, filter, split, and take. You may need to use an if statement to implement filter. The syntax for this is

  ```
  ```

- Exercise: ListsAndRecursion-6

  (Hard) Write a flatten(list) function that takes a list that may contain any number of sublists, which themselves may contain sublists, to any depth. It returns the elements of these lists as a flat list.

  ```
  ```

  Hint: You may have to use Enum.reverse to get your result in the correct order.

[illegible]map(&(&1*&1))

Streams—Lazy Enumerables

In Elixir, the Enum module is greedy. This means that when you pass it a collection, it potentially consumes all the contents of that collection. It also means the result will typically be another collection. Look at the following pipeline:

enum/pipeline.exs

The first map function takes the original list and creates a new list of its squares. with_index takes this list and returns a list of tuples. The next map then subtracts the index from the value, generating a list that gets passed to

IO.inspect.

So, this pipeline generates four lists on its way to outputting the final result.

Let's look at something different. Here's some code that reads lines from a file and returns the longest:

enum/longest_line.exs

In this case, we read the whole dictionary into memory (on my machine that's 2.4MB), then split it into a list of words (236,000 of them) before processing it to find the longest (which happens to be *formaldehydesulphoxylate*).

In both of these examples, our code is

suboptimal because each call to Enum is self-contained. Each call takes a collection and returns a collection.

What we really want is to process the elements in the collection as we need them. We don't need to store intermediate results as full collections; we just need to pass the current element from function to function. And that's what streams do.

A Stream Is a Composable Enumerator

Here's a simple example of creating a Stream:

If we'd called Enum.map, we'd have seen the result [2,4,6,8] come back

immediately. Instead we get back a stream value that contains a specification of what we intended.

How do we get the stream to start giving us results? Treat it as a collection and pass it to a function in the Enum module:

Because streams are enumerable, you can also pass a stream to a stream function. Because of this, we say that streams are *composable*.

Of course, in real life we'd have written this as

enum/stream1.exs

Note that we're never creating intermediate lists—we're just passing successive elements of each of the collections to the next in the chain. The Stream values shown in the previous IEx session give a hint of how this works—chained streams are represented as a list of functions, each of which is applied in turn to each element of the stream as it is processed.

Streams aren't only for lists. More and more Elixir modules now support streams. For example, here's our longest-word code written using streams:

enum/stream2.exs

The magic here is the call to IO.stream, which converts an IO device (in this case the open file) into a stream that serves one line at a time. In fact, this is such a useful concept that there's a shortcut:

enum/stream3.exs

The good news is that there is no intermediate storage. The bad news is that it runs about two times slower than the previous version. However, consider the case where we were reading data from a remote server or from an external sensor (maybe temperature readings). Successive lines might arrive slowly, and they might go on for ever. With the Enum implementation we'd have to wait

for all the lines to arrive before we started processing. With streams we can process them as they arrive.

Infinite Streams

Because streams are lazy, there's no need for the whole collection to be available up front. For example, if I write

it takes about 8 seconds before I see the result. Elixir is creating a 10-million-element list, then taking the first five elements from it. If instead I write

the result comes back instantaneously. The take call just needs five values,

which it gets from the stream. Once it has them, there's no more processing.

In these examples the stream is bounded, but it can equally well go on forever. To do that, we'll need to create streams based on functions.

Creating Your Own Streams

Streams are implemented solely in Elixir libraries—there is no specific runtime support. However, this doesn't mean you want to drop down to the very lowest level and create your own streamable types. The actual implementation is complex (in the same way that string theory and dating rituals are complex). Instead, you probably want to use some helpful wrapper functions to do the heavy lifting. There

are a number of these, including cycle, repeatedly, iterate, unfold, and resource. (If you needed proof that the internal implementation is tricky, consider the fact that these last two names give you almost no hint of their power.)

Let's start with the three simplest: cycle, repeatedly, and iterate.

Stream.cycle

Stream.cycle takes an enumerable and returns an infinite stream containing that enumerable's elements. When it gets to the end, it repeats from the beginning, indefinitely. Here's an example that generates the rows in an HTML table with alternating *green* and *white* classes:

Stream.repeatedly

Stream.repeatedly takes a function and invokes it each time a new value is wanted.

Stream.iterate

Stream.iterate(start_value, next_fun) generates an infinite stream. The first value is start_value. The next value is generated by applying next_fun to this value. This continues for as long as the stream is being used, with each value being the result of applying next_fun to the previous value.

Here are some examples:

Stream.unfold

Now we can get a little more adventurous. Stream.unfold is related to iterate, but you can be more explicit both about the values output to the stream and about the values passed to the next iteration. You supply an initial value and a function. The function uses the argument to create two values, returned as a tuple. The first is the value to be returned by this iteration of the stream, and the second is the value to be passed to the function on the next iteration of the stream. If the function returns nil, the stream terminates.

This sounds abstract, but unfold is quite useful—it is a general way of creating a potentially infinite stream of values where each value is some function of

the previous state.

The key is the generating function. Its general form is

For example, here's a stream of Fibonacci numbers:

Here the *state* is a tuple containing the current and the next number in the sequence. We seed it with the initial state of {0, 1}. The value each iteration of the stream returns is the first of the state values. The new state moves one down the sequence, so an initial state of {f1,f2} becomes a new state of {f2,f1 + f2}.

Stream.resource

At this point you might be wondering how streams can interact with external resources. We've already seen how you can turn a file's contents into a stream of lines, but how could you implement this yourself? You'd need to open the file when the stream first starts, return successive lines, and then close the file at the end. Or maybe you want to turn a database result-set cursor into a stream of values. You'd have to execute the query when the stream starts, return each row as stream values, and close the query at the end. And that's where Stream.resource comes in.

Stream.resource builds upon Stream.unfold. It makes two changes.

The first argument to unfold is the initial value to be passed to the iteration function. But if that value is a resource, we don't want to open it until the stream starts delivering values, and that might not happen until long after we create the stream. To get around this, resource takes not a value, but a function that returns the value. That's the first change.

Second, when the stream is done with the resource, we may need to close it. That's what the third argument to Stream.resource does—it takes the final accumulator value and does whatever is needed to deallocate the resource.

Here's an example from the library documentation:

The first function opens the file when the stream becomes active, and passes it to the second function. This reads the file, line by line, returning either a line and the file as a tuple, or a :halt tuple at the end of the file. The third function closes the file.

Let's finish with a different kind of resource: time. We'll implement a timer that counts down the number of seconds until the start of the next minute. It uses a stream resource to do this. The allocation function returns the number of seconds left until the next minute starts. It does this each time the stream is evaluated, so we'll get a countdown that varies depending on when it is called.

The iteration function looks at the time left. If zero, it returns {:halt, 0}; otherwise it sleeps for a second and returns the current countdown as a string, along with the decremented counter.

In this case there's no resource deallocation, so the third function does nothing.

Here's the code:

enum/countdown.exs

(The eagle-eyed among you will have noticed a function called say in the Countdown module. This executes the

shell command say, which, on OS X, speaks its argument. You could substitute espeak on Linux and ptts on Windows.)

Let's play with the code.

So far, we've built a stream that creates time events, prints the countdown value, and speaks it. But there's been no output, as we haven't yet asked the stream for any values. Let's do that now:

Cool—we must have started it around 22 seconds into a minute, so the countdown starts at 37. Let's use the

same stream again, a few seconds later:

Wait some more seconds, and this time let it run to the top of the minute:

This is clearly not great code, as it fails to correct the sleep time for any delays introduced by our code. But it illustrates a very cool point. Lazy streams let you deal with resources that are asynchronous to your code, and the fact that they are initialized every time they are used means they're effectively side effect–free. Every time we pipe our stream to an Enum function, we get a fresh set of values, computed at that time.

Streams in Practice

In the same way that functional programming requires you to look at problems in a new way, streams ask you to look at iteration and collections afresh. Not every situation where you're iterating requires a stream. But consider using a stream when you want to defer processing until you need the data, and when you need to deal with large numbers of things without necessarily generating them all at once.

The Collectable Protocol

The Enumerable protocol lets you iterate over the elements in a type—given a collection, you can get the elements. Collectable is in some sense the opposite—it allows you to build a collection by inserting elements into it.

Not all collections are collectable. Ranges, for example, cannot have new entries added to them.

The collectable API is pretty low-level, so you'll typically access it via Enum.into and when using comprehensions (which we cover in the next section). For example, we can

inject the elements of a range into an empty list using

If the list is not empty, the new elements are tacked onto the end:

Output streams are collectable, so the following code lazily copies standard input to standard output:

Comprehensions

When you're writing functional code, you often map and filter collections of things. To make your life easier (and your code easier to read), Elixir provides a general-purpose shortcut for this: the *comprehension*.

The idea of a comprehension is fairly simple: given one or more collections, extract all combinations of values from each, optionally filter the values, and then generate a new collection using the values that remain.

The general syntax for comprehensions is deceptively simple:

result = for *generator or filter…* [, into: *value*], do: *expression*

Let's see a couple of basic examples before we get into the details.

A generator specifies how you want to extract values from a collection.

Any variables matched in the pattern are available in the rest of the comprehension (including the block). For example, `x <- [1,2,3]` says that we want to first run the rest of the comprehension with `x` set to 1. Then we run it with `x` set to 2, and so on. If we have two generators, their operations are

nested, so

will run the rest of the comprehension with `x = 1, y = 5`; `x = 1, y = 6`; `x = 2, y = 5`; and `x = 2, y = 6`. We can use those values of `x` and `y` in the `do` block:

You can use variables from generators in later generators:

A filter is a predicate. It acts as a gatekeeper for the rest of the comprehension—if the condition is false, then the comprehension moves on to the next iteration without generating an output value.

For example, the code that follows uses a comprehension to list pairs of numbers from 1 to 8 whose product is a multiple of 10. It uses two generators (to cycle through the pairs of numbers) and two filters. The first filter allows only pairs in which the first number is at least the value of the second. The second filter checks to see if the product is a multiple of 10.

This comprehension iterates 64 times, with `x=1`, `y=1`; `x=1`, `y=2`; and so on. However, the first filter cuts the iteration short when `x` is less than `y`. This means the second filter runs only 36 times.

Because the first term in a generator is a

pattern, we can use it to deconstruct structured data. Here's a comprehension that swaps the keys and values in a keyword list.

Comprehensions Work on Bits, Too

A bitstring (and, by extension, a binary or a string) is simply a collection of ones and zeroes. So it's probably no surprise that comprehensions work on bits, too. What might be surprising is the syntax:

Here the generator is enclosed in `<<` and `>>`, indicating a binary. In the first case, the `do` block returns the integer code for each character, so the resulting list is `[104, 101, 108, 108, 111]`, which

IEx displays as 'hello'.

In the second case, we convert the code back into a string, and the result is a list of those one-character strings.

Again, the thing to the left of the `<-` is a pattern, and so we can use binary pattern matching. Let's convert a string into the octal representation of its characters:

Scoping and Comprehensions

All variable assignments inside a comprehension are local to that comprehension—you will not affect the value of a variable in the outer scope.

The Value Returned by a Comprehension

In our examples thus far, the comprehension has returned a list. The list contains the values returned by the do expression for each iteration of the comprehension.

This behavior can be changed with the into: parameter. This takes a collection that is to receive the results of the comprehension. For example, we can populate a map using

It might be more clear to use Map.new in this case:

The collection doesn't have to be empty:

In Chapter 24, *Protocols—Polymorphic Functions*, we'll look at protocols, which let us specify common behaviors across different types. The into: option takes values that implement the Collectable protocol. These include lists, binaries, functions, maps, files, hash dicts, hash sets, and IO streams, so we can write things such as

[illegible]unt:

Your Turn

- Exercise: ListsAndRecursion-7

  In the last exercise of Chapter 7, *Lists and Recursion*, you wrote a span function. Use it and list comprehensions to return a list of the prime numbers from 2 to *n*.

- Exercise: ListsAndRecursion-8

  The Pragmatic Bookshelf has offices in Texas (TX) and North Carolina (NC), so we have to charge sales tax on orders shipped to these states. The rates can be expressed as a keyword list (I wish it were that simple.…):

  ```
  ```

  Here's a list of orders:

  ```
  ```

  Write a function that takes both lists and returns a copy of the orders, but with an extra field, total_amount, which is the net plus sales tax. If a shipment is not to NC or TX, there's no tax applied.

Moving Past Divinity

L. Peter Deutsch once penned, “To iterate is human, to recurse divine.” And that’s certainly the way I felt when I first started coding Elixir. The joy of pattern-matching lists in sets of recursive functions drove my designs. After a while, I realized that perhaps I was taking this too far.

In reality, most of our day-to-day work is better handled using the various enumerators built into Elixir. They make your code smaller, easier to understand, and probably more efficient.

Part of the process of learning to be

effective in Elixir is working out for yourself when to use recursion and when to use enumerators. I recommend enumerating when you can.

Next we'll look at string handling in Elixir (and Erlang).

Footnotes

[18] http://elixir-lang.org/docs/

Copyright © 2018, The Pragmatic Bookshelf.

We cover:

- *Strings and string literals*
- *Character lists (single-quoted literals)*
- *Pattern matching and processing strings*

Chapter 11

Strings and Binaries

We've been using strings without really discussing them. Let's rectify that.

String Literals

Elixir has two kinds of string: single-quoted and double-quoted. They differ significantly in their internal representation. But they also have many things in common.

- Strings can hold characters in UTF-8 encoding.
- They may contain escape sequences:

- They allow interpolation on Elixir expressions using the syntax `#{...}`:

- Characters that would otherwise have special meaning can be escaped with a backslash.
- They support *heredocs*.

Heredocs

Any string can span several lines. To illustrate this, we'll use both IO.puts and IO.write. We use write for the multiline string because puts always appends a newline, and we want to see the contents without this.

produces

Notice how the multiline string retains the leading and trailing newlines and the leading spaces on the intermediate lines.

The *heredoc* notation fixes this. Triple the string delimiter (''' or """) and indent the trailing delimiter to the same margin

as your string contents, and you get this:

which produces

Heredocs are used extensively to add documentation to functions and modules.

Sigils

Like Ruby, Elixir has an alternative syntax for some literals. We've already seen it with regular expressions, where we wrote `~r{...}`. In Elixir, these ~-style literals are called *sigils* (symbols with magical powers).

A sigil starts with a tilde, followed by an

upper- or lowercase letter, some delimited content, and perhaps some options. The delimiters can be <...>, {...}, [...], (...), |...|, /.../, "...", and '...'.

The letter determines the sigil's type:

Here are some examples of sigils, using a variety of delimiters:

The ~W and ~w sigils take an optional type specifier, a, c, or s, which determines whether it returns a list of atoms, character lists, or strings. (We've already seen the ~r options.)

The delimiter can be any nonword character. If it is (, [, {, or <, then the terminating delimiter is the corresponding closing character. Otherwise the terminating delimiter is the next nonescaped occurrence of the opening delimiter.

Elixir does not check the nesting of delimiters, so the sigil `~s{a{b}` is the three-character string `a{b`.

If the opening delimiter is three single or three double quotes, the sigil is treated as a heredoc.

If you want to specify modifiers with heredoc sigils (most commonly you'd

do this with ~r), add them after the trailing delimiter.

One of the interesting things about sigils is that you can define your own. We talk about this in Part III,.

The Name “strings”

Before we get further into this, I need to explain something. In most other languages, you’d call both ’cat’ and "cat" *strings*. And that’s what I’ve been doing so far. But Elixir has a different convention.

In Elixir, the convention is that we call only double-quoted strings “strings.” The single-quoted form is a character list.

This is important. The single- and double-quoted forms are very different, and libraries that work on strings work only on the double-quoted form.

Let’s explore the differences in more detail.

Single-Quoted Strings—Lists of Character Codes

Single-quoted strings are represented as a list of integer values, each value corresponding to a codepoint in the string. For this reason, we refer to them as *character lists* (or *char lists*).

This is confusing: IEx *says* it is a list, but it shows the value as a string. That's because IEx prints a list of integers as a string if it believes each number in the list is a printable character. You can try this for yourself:

You can look at the internal representation in a number of ways:

The ~w in the format string forces str to be written as an Erlang term—the underlying list of integers. The ~n is a newline.

The last example creates a new character list with a null byte at the end. IEx no longer thinks all the bytes are printable, and so returns the underlying character codes.

If a character list contains characters Erlang considers nonprintable, you'll see the list representation.

Because a character list is a list, we can use the usual pattern matching and List functions.

Why is the head of 'cat' 99 and not *c*?. Remember that a char list is just a list of integer character codes, so each individual entry is a number. It happens that 99 is the code for a lowercase *c*.

In fact, the notation ?c returns the integer code for the character c. This is often useful when employing patterns to extract information from character lists. Here's a simple module that parses the character-list representation of an optionally signed decimal number.

strings/parse.exs

Let’s try it in IEx.

[100,111,103]

Your Turn

- Exercise: StringsAndBinaries-1

  Write a function that returns true if a single-quoted string contains only printable ASCII characters (space through tilde).

- Exercise: StringsAndBinaries-2

  Write an anagram?(word1, word2) that returns true if its parameters are anagrams.

- Exercise: StringsAndBinaries-3

  Try the following in IEx:

  ```
  ```

  Why does IEx print 'cat' as a string, but 'dog' as individual numbers?

- Exercise: StringsAndBinaries-4

  (Hard) Write a function that takes a single-quoted string of the form *number* [+-*/] *number* and returns the result of the calculation. The individual numbers do not

have leading plus or minus signs.

```
calculate('123 + 27') # => 150
```

Binaries

The binary type represents a sequence of bits.

A binary literal looks like << *term,…* >>.

The simplest term is just a number from 0 to 255. The numbers are stored as successive bytes in the binary.

You can specify modifiers to set any term's size (in bits). This is useful when working with binary formats such as media files and network packets.

You can store integers, floats, and other binaries in binaries.

Let's finish an initial look at binaries with an example of bit extraction. An IEEE 754 float has a sign bit, 11 bits of exponent, and 52 bits of mantissa. The exponent is biased by 1023, and the mantissa is a fraction with the top bit assumed to be 1. So we can extract the fields and then use :math.pow, which performs exponentiation, to reassemble the number:

[illegible]codepoints(dqs)

Double-Quoted Strings Are Binaries

Whereas single-quoted strings are stored as char lists, the contents of a double-quoted string (*dqs*) are stored as a consecutive sequence of bytes in UTF-8 encoding. Clearly this is more efficient in terms of memory and certain forms of access, but it does have two implications.

First, because UTF-8 characters can take more than a single byte to represent, the size of the binary is not necessarily the length of the string.

Second, because you're no longer using

lists, you need to learn and work with the binary syntax alongside the list syntax in your code.

Strings and Elixir Libraries

When Elixir library documentation uses the word *string* (and most of the time it uses the word *binary*), it means double-quoted strings.

The String module defines functions that work with double-quoted strings.

> *at(str, offset)*
>
> Returns the grapheme at the given offset (starting at 0). Negative offsets count from the end of the string.

capitalize(str)

Converts str to lowercase, and then capitalizes the first character.

codepoints(str)

Returns the codepoints in str.

downcase(str)

Converts str to lowercase.

duplicate(str, n)

Returns a string containing n copies of str.

ends_with?(str, suffix | [suffixes])

Returns true if str ends with any of the given suffixes.

first(str)

Returns the first grapheme from str.

graphemes(str)

Returns the graphemes in the string. This is different from the codepoints function, which lists combining characters separately. The following example uses a combining diaeresis along with the letter *e* to represent *ë*. (It might not display properly on

your ereader.)

jaro_distance

Returns a float between 0 and 1 indicating the likely similarity of two strings.

last(str)

Returns the last grapheme from str.

length(str)

Returns the number of graphemes in str.

myers_difference

Returns the list of transformations needed to convert one string to another.

next_codepoint(str)

Splits str into its leading codepoint and the rest, or nil if str is empty. This may be used as the basis of an iterator.

produces

next_grapheme(str)

Same as next_codepoint, but

returns graphemes (:no_grapheme on completion).

pad_leading(str, new_length, padding \\ " ")

Returns a new string, at least new_length characters long, containing str right-justified and padded with padding.

pad_trailing(str, new_length, padding \\ " ")

Returns a new string, at least new_length characters long, containing str left-justified and padded with padding.

printable?(str)

Returns true if str contains only

printable characters.

replace(str, pattern, replacement, options \\ [global: true, insert_replaced: nil])

Replaces pattern with replacement in str under control of options.

If the :global option is true, all occurrences of the pattern are replaced; otherwise only the first is replaced.

If :insert_replaced is a number, the pattern is inserted into the replacement at that offset. If the option is a list, it is inserted multiple times.

reverse(str)

Reverses the graphemes in a string.

slice(str, offset, len)

Returns a len character substring starting at offset (measured from the end of str if negative).

split(str, pattern \\ nil, options \\ [global: true])

Splits str into substrings delimited by pattern. If :global is false, only one split is performed. pattern can be a string, a regular expression, or nil. In the latter case, the string is split on

whitespace.

```
```

starts_with?(str, prefix | [prefixes])

Returns true if str starts with any of the given prefixes.

```
```

trim(str)

Trims leading and trailing whitespace from str.

```
```

trim(str, character)

Trims leading and trailing instances of character from str.

```
```

trim_leading(str)

Trims leading whitespace from str.

trim_leading(str, character)

Trims leading copies of character (an integer codepoint) from str.

trim_trailing(str)

Trims trailing whitespace from str.

trim_trailing(str, character)

Trims trailing occurrences of character from str.

upcase(str)

valid?(str)

Returns true if str is a string containing valid codepoints.

Your Turn

- Exercise: StringsAndBinaries-5

  Write a function that takes a list of double-quoted strings and prints each on a separate line, centered in a column that has the width of the longest string. Make sure it works with UTF characters.

integer-signed-

Binaries and Pattern Matching

The first rule of binaries is “if in doubt, specify the type of each field.” Available types are binary, bits, bitstring, bytes, float, integer, utf8, utf16, and utf32. You can also add qualifiers:

- size(*n*): The size of the field, in bits.
- signed or unsigned: For integer fields, should it be interpreted as signed?
- endianness: big, little, or native.

Use hyphens to separate multiple attributes for a field:

```
```

However, unless you’re doing a lot of

work with binary file or protocol formats, the most common use of all this scary stuff is to process UTF-8 strings.

String Processing with Binaries

When we process lists, we use patterns that split the head from the rest of the list. With binaries that hold strings, we can do the same kind of trick. We have to specify the type of the head (UTF-8), and make sure the tail remains a binary.

strings/utf-iterate.ex

produces

The parallels with list processing are clear, but the differences are significant.

Rather than use `[ head | tail ]`, we use `<< head::utf8, tail::binary >>`. And rather than terminate when we reach the empty list, `[]`, we look for an empty binary, `<<>>`.

Your Turn

- Exercise: StringsAndBinaries-6

  Write a function to capitalize the sentences in a string. Each sentence is terminated by a period and a space. Right now, the case of the characters in the string is random.

- Exercise: StringsAndBinaries-7

  Chapter 7 had an exercise about calculating sales tax. We now have the sales information in a file of comma-separated id, ship_to, and amount values. The file looks like this:

  Write a function that reads and parses this file and then passes the result to the sales_tax function. Remember that the data should be formatted into a keyword list, and that the fields need to be the correct types (so the id field is an integer, and so on).

You'll need the library functions `File.open`, `IO.read(file, :line)`, and `IO.stream(file)`.

Familiar Yet Strange

String handling in Elixir is the result of a long evolutionary process in the underlying Erlang environment. If we were starting from scratch, things would probably look a little different. But once you get over the slightly strange way that strings are matched using binaries, you'll find that it works out well. In particular, pattern matching makes it very easy to look to strings that start with a particular sequence, which in turn makes simple parsing tasks a pleasure to write.

You may have noticed that we're a long way into the book and haven't yet talked

about control-flow constructs such as if and case. This is deliberate: we use them less often in Elixir than in more conventional languages. However, we still need them, so they are the subject of the next chapter.

Copyright © 2018, The Pragmatic Bookshelf.

We cover:

- *if and unless*
- *cond (a multiway if)*
- *case (a pattern-matching switch)*
- *Exceptions*

Chapter 12

Control Flow

Elixir code tries to be declarative, not imperative.

In Elixir we write lots of small functions, and a combination of guard clauses and pattern matching of parameters replaces most of the control

flow seen in other languages.

However, Elixir does have a small set of control-flow constructs. The reason I've waited so long to introduce them is that I want you to try not to use them much. You definitely will, and should, drop the occasional cond or case into your code. But before you do, consider more functional alternatives. The benefit will become obvious as you write more code—functions written without explicit control flow tend to be shorter and more focused. They're easier to read, test, and reuse. If you end up with a 10- or 20-line function in an Elixir program, it is pretty much guaranteed that it will contain one of the constructs in this chapter and that you can simplify it.

So, forewarned, let’s go.

if and unless

In Elixir, if and its evil twin, unless, take two parameters: a condition and a keyword list, which can contain the keys do: and else:. If the condition is truthy, the if expression evaluates the code associated with the do: key; otherwise it evaluates the else: code. The else: branch may be absent.

Just as it does with function definitions, Elixir provides some syntactic sugar. You can write the first of the previous examples as follows:

unless is similar:

The value of if and unless is the value of the expression that was evaluated.

cond

The cond macro lets you list out a series of conditions, each with associated code. It executes the code corresponding to the first truthy conditions.

In the game of FizzBuzz, children count up from 1. If the number is a multiple of three, they say “Fizz.” For multiples of five, they say “Buzz.” For multiples of both, they say “FizzBuzz.” Otherwise, they say the number.

In Elixir, we could code this as follows:

control/fizzbuzz.ex

The cond starts on line 8. We assign the value of the cond expression to next_answer. Inside the cond, we have four alternatives—the current number is a multiple of 3 and 5, just 3, just 5, or neither. Elixir examines each in turn and returns the value of the expression following the -> for the first true one. The _upto function then recurses to find the next value. Note the use of true -> to handle the case where none of the previous conditions match. This is the equivalent of the else or default stanza of a more traditional case statement.

There's a minor problem, though. The result list we build always has the most recent value as its head. When we finish, we'll end up with a list that has the

answers in reverse order. That's why in the anchor case (when left is zero), we reverse the result before returning it. This is a very common pattern. And don't worry about performance—list reversal is highly optimized.

Let's try the code in IEx:

In this case, we could do something different and remove the call to reverse. If we process the numbers in reverse order (so we start at n and end at 1), the resulting list will be in the correct order.

control/fizzbuzz1.ex

This code is quite a bit cleaner than the

previous version. However, it is also slightly less idiomatic—readers will expect to traverse the numbers in a natural order and reverse the result.

There's a third option. FizzBuzz transforms a number into a string. We like to code things as transformations, so let's use Enum.map to transform the range of numbers from 1 to n to the corresponding FizzBuzz words.

control/fizzbuzz2.ex

This section is intended to show you how cond works, but you'll often find that it's better not to use it, and instead to take advantage of pattern matching in function calls. The choice is yours.

control/fizzbuzz3.ex

case

case lets you test a value against a set of patterns, executes the code associated with the first pattern that matches, and returns the value of that code. The patterns may include guard clauses.

For example, the File.open function returns a two-element tuple. If the open is successful, it returns {:ok, file}, where file is an identifier for the open file. If the open fails, it returns {:error, reason}. We can use case to take the appropriate action when we open a file. (Here, the code opens its own source file.)

control/case.ex

produces

If we change the file name to something that doesn't exist and then rerun the code, we instead get Failed to open file: enoent.

We can use the full power of nested pattern matches:

control/case1.exs

We've seen how to employ guard clauses to refine the pattern used when matching functions. We can do the same with case.

control/case2.exs

Raising Exceptions

First, the official warning: exceptions in Elixir are *not* control-flow structures. Instead, Elixir exceptions are intended for things that should never happen in normal operation. That means the database going down or a name server failing to respond could be considered exceptional. Failing to open a configuration file whose name is fixed could be seen as exceptional. However, failing to open a file whose name a user entered is not. (You could anticipate that a user might mistype it every now and then.)

Raise an exception with the raise

function. At its simplest, you pass it a string and it generates an exception of type RuntimeError.

You can also pass the type of the exception, along with other optional attributes. All exceptions implement at least the message attribute.

You use exceptions far less in Elixir than in other languages—the design philosophy is that errors should propagate back up to an external, supervising process. We'll cover this when we talk about OTP supervisors.

Elixir has all the usual exception-

catching mechanisms. To emphasize how little you should use them, I've described them in an appendix.

(file_name)

Designing with Exceptions

If File.open succeeds, it returns {:ok, file}, where file is the service that gives you access to the file. If it fails, it returns {:error, reason}. So, for code that knows a file open might not succeed and wants to handle the fact, you might write

If instead you expect the file to open successfully every time, you could raise an exception on failure.

Or you could let Elixir raise an exception for you and write

If the pattern match on the first line fails, Elixir will raise a MatchError exception. It won't be as informative as our version that handled the error explicitly, but if the error should never happen, this form is probably good enough (at least until it triggers the first time and the operations folks say they'd like more information).

An even better way to handle this is to use File.open!. The trailing exclamation point in the method name is an Elixir convention—if you see it, you know the function will raise an exception on error, and that exception will be meaningful. So we could simply write

and get on with our lives.

Doing More with Less

Elixir has just a few forms of control flow: if, unless, cond, case, and (perhaps) raise. But surprisingly, this doesn't matter in practice. Elixir programs are rich and expressive without a lot of branching code. And they're easier to work with as a result.

That concludes our basic tour of Elixir. Now let's start putting it all together and implement a full project.

Your Turn

- Exercise: ControlFlow-1

  Rewrite the FizzBuzz example using case.

- Exercise: ControlFlow-2

  We now have three different implementations of FizzBuzz. One uses cond, one uses case, and one uses separate functions with guard clauses.

  Take a minute to look at all three. Which do you feel best expresses the problem. Which will be easiest to maintain?

  The case style and the implementation using guard clauses are different from control structures in most other languages. If you feel that one of these was the best implementation, can you think of ways to remind yourself to investigate these options as you write Elixir code in the future?

- Exercise: ControlFlow-3

  Many built-in functions have two forms. The

xxx form returns the tuple {:ok, data} and the *xxx!* form returns data on success but raises an exception otherwise. However, some functions don't have the *xxx!* form.

Write an ok! function that takes an arbitrary parameter. If the parameter is the tuple {:ok, data}, return the data. Otherwise, raise an exception containing information from the parameter.

You could use your function like this:

Copyright © 2018, The Pragmatic Bookshelf.

We cover:

- *Project structure*
- *The mix build tool*
- *ExUnit testing framework*
- *Documentation*

Chapter 13

Organizing a Project

Let’s stop hacking and get serious.

You’ll want to organize your source code, write tests, and handle any dependencies. And you’ll want to follow Elixir conventions, because that way you’ll get support from the tools.

In this chapter we'll look at mix, the Elixir build tool. We'll investigate the directory structure it uses and see how to manage external dependencies. And we'll end up using ExUnit to write tests for our code (and to validate the examples in our code's documentation). To motivate this, we'll write a tool that downloads and lists the *n* oldest issues from a GitHub project. Along the way, we'll need to find some libraries and make some design decisions typical of an Elixir project. We'll call our project issues.

[illegible]

The Project: Fetch Issues from GitHub

GitHub provides a nice web API for fetching issues.[19] Simply issue a GET request to

and you'll get back a JSON list of issues. We'll reformat this, sort it, and filter out the oldest *n*, presenting the result as a table:

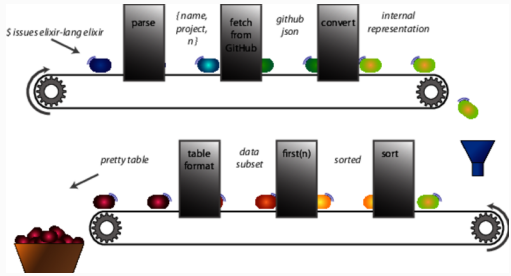
How Our Code Will Do It

Our program will run from the command line. We'll need to pass in a GitHub user name, a project name, and

an optional count. This means we'll need some basic command-line parsing.

We'll need to access GitHub as an HTTP client, so we'll have to find a library that gives us the client side of HTTP. The response that comes back will be in JSON, so we'll need a library that handles JSON, too. We'll need to be able to sort the resulting structure. And finally, we'll need to lay out selected fields in a table.

We can think of this data transformation in terms of a production line. Raw data enters at one end and is transformed by each of the stations in turn.



Here we see data, starting at the command line and ending at *pretty table*. At each stage, it undergoes a transformation (parse, fetch, and so on). These transformations are the functions we write. We'll cover each one in turn.

Step 1: Use Mix to Create Our New Project

Mix is a command-line utility that manages Elixir projects. Use it to create new projects, manage a project's dependencies, run tests, and run your code. If you have Elixir installed, you also have mix. Try running it now:

This is a list of the standard tasks that come with mix. (Your list may be a little different, depending on your version of Elixir.) For more information on a particular task, use mix help *taskname*.

You can write your own mix tasks, for a project and to share between projects.[20]

Create the Project Tree

Each Elixir project lives in its own directory tree. If you use mix to manage this tree, then you'll follow the mix conventions (which are also the conventions of the Elixir community). We'll use these conventions in the rest of this chapter.

We'll call our project issues, so it will go in a directory named issues. We'll create this directory using mix.

At the command line, navigate to a place where you want this new project to live, and type

In tree form, the newly created files and directories look like this:

Change into the issues/ directory. This is a good time to set up version control. I use Git, so I do

(I don't want to clutter the book with version-control stuff, so that's the last time I'll mention it. Make sure you follow your own version-control practices as we go along.)

Our new project contains three directories and some files.

.formatter.exs

Configuration used by the source code formatter

.gitignore

Lists the files and directories generated as by-products of the build and not to be saved in the repository.

README.md

A place to put a description of your project (in Markdown format). If you store your project on GitHub, this file's contents will appear on the project's home page.

config/

Eventually we'll put some application-specific configuration

here.

lib/

This is where our project's source lives. Mix has already added a top-level module (issues.ex in our case).

mix.exs

This source file contains our project's configuration options. We will be adding stuff to this as our project progresses.

test/

A place to store our tests. Mix has already created a helper file and a stub for unit tests of the issues module.

Now our job is to add our code. But before we do, let’s think a little about the implementation.

Transformation: Parse the Command Line

Let's start with the command line. We really don't want to couple the handling of command-line options into the main body of our program, so let's write a separate module to interface between what the user types and what our program does. By convention this module is called *Project*.CLI (so our code would be in Issues.CLI). Also by convention, the main entry point to this module will be a function called run that takes an array of command-line arguments.

Where should we put this module?

Elixir has a convention. Inside the lib/ directory, create a subdirectory with the same name as the project (so we'd create the directory lib/issues/). This directory will contain the main source for our application, one module per file. And each module will be namespaced inside the Issues module—the module naming follows the directory naming.

In this case, the module we want to write is Issues.CLI—it is the CLI module nested inside the Issues module. Let's reflect that in the directory structure and put cli.ex in the lib/issues directory:

Elixir comes bundled with an option-parsing library,[21] so we will use that.

We'll tell it that -h and --help are possible switches, and anything else is an argument. It returns a tuple, where the first element is a keyword list of the options and the second is a list of the remaining arguments. Our initial CLI module looks like the following:

project/0/issues/lib/issues/cli.ex

Write Some Basic Tests

At this point, I get a little nervous if I don't have some tests. Fortunately, Elixir comes with a wonderful (and simple) testing framework called ExUnit.

Have a look at the file test/issues_test.exs.

project/0/issues/test/issues_test.exs

It acts as a template for all the test files you write. I just copy and paste the boilerplate into separate test files as I need them. So let's write tests for our CLI module, putting those tests into the

file test/cli_test.exs. (Test file names must end with _test.) We'll test that the option parser successfully detects the -h and --help options, and that it returns the arguments otherwise. We'll also check that it supplies a default value for the count if only two arguments are given.

project/1/issues/test/cli_test.exs

These tests all use the basic assert macro that ExUnit provides. This macro is clever—if an assertion fails, it can extract the values from the expression you pass it, giving you a nice error message.

To run our tests, we'll use the mix test task.

One of the four tests failed. When we pass a count as the third parameter, our code blows up. See how the assertion shows you its type (`==` in this case), the line of code that failed, and the two values that we compared. You can see the difference between the left-hand side (lhs), which is the value returned by `parse_args`, and the expected value (the rhs)—if your terminal and your eyes support it, you'll see that the `"99"` in the line labelled `lhs:` is colored red, and the `99` in the next line is green. We were expecting to get a number as the count, but we got a string.

That's easily fixed. The built-in function `String.to_integer` converts a binary (a

string) into an integer.

project/1/issues/lib/issues/cli.ex

Your Turn

- Exercise: OrganizingAProject-1

  Do what I did. Honest. Create the project and write and test the option parser. It's one thing to read about it, but you'll be doing this a lot, so you may as well start now.

[illegible]

Refactor: Big Function Alert

Our parse_args function is waving two red flags. First, it contains conditional logic. Second, it is too long. Let's split it up.

project/1a/issues/lib/issues/cli.ex

And run the tests:

parse_args(argv))

Transformation: Fetch from GitHub

Now let's continue down our data-transformation chain. Having parsed our arguments, we need to transform them by fetching data from GitHub. So we'll extend our run function to call a process function, passing it the value returned from the parse_args function. We could have written this:

But to understand this code, you have to read it right to left. I prefer to make the chain more explicit using the Elixir pipe operator:

project/1a/issues/lib/issues/cli.ex

We need two variants of the process function. One handles the case where the user asked for help and parse_args returned :help. The other handles the case where a user, a project, and a count are returned.

project/1a/issues/lib/issues/cli.ex

We can use mix to run our function. Let's first see if help gets displayed.

You pass mix run an Elixir expression, which gets evaluated in the context of your application. Mix will recompile your application, as it is out of date, before executing the expression.

If we pass it user and project names, however, it'll blow up because we haven't written that code yet.

Let's write that code now. Our program will act as an HTTP client, accessing GitHub through its web API. So, it looks like we'll need an external library.

Step 2: Use Libraries

Elixir comes with a bunch of libraries preinstalled. Some are written in Elixir, and others in Erlang.

The first port of call is http://elixir-lang.org/docs.html, the Elixir documentation. Often you'll find a built-in library that does what you want.

Next, see if any standard Erlang libraries do what you need. This isn't a simple task. Visit http://erlang.org/doc/ and look in the left sidebar for *Application Groups*. There you'll find libraries sorted by top-level category.

If you find what you're looking for in

either of these places, you're all set, as all these libraries are already available to your application. But if the built-in libraries don't contain what you need, you'll have to add an external dependency.

Finding an External Library

Package managers: Ruby has RubyGems, Python has pip, Node.js has npm.

And Elixir has *hex*.

Visit https://hex.pm and search its list of packages that integrate nicely with a mix-based project.

If all else fails, Google and GitHub are your friends. Search for terms such as

elixir http client or *erlang distributed logger*, and you're likely to turn up the libraries you need.

In our case, we need an HTTP client. We find that Elixir has nothing built in, but hex.pm has a number of HTTP client libraries.

To me, HTTPoison looks like a good option. So how do we include it in our project?

Adding a Library to Your Project

Mix takes the view that all external libraries should be copied into the project's directory structure. The good news is that it handles all this for us—we just need to list the dependencies, and it does the rest. Remember the

mix.exs file at the top level of our project? Here is that original version.

project/0/issues/mix.exs

We add new dependencies to the deps function. As the HTTPoison package is in hex.pm, that's very simple. We just give the name and the version we want.

project/1a/issues/mix.exs

In this case, we give the version as "~> 1.0.0". This matches any version of HTTPoison with a major version of 1 and a minor version of 0 or greater. In IEx, type h Version for more details.

Once your mix.exs file is updated,

you're ready to have mix manage your dependencies.

Use mix deps to list the dependencies and their status:

Download the dependencies with mix deps.get:

Run mix deps again:

This shows that the HTTPoison library is installed but that it hasn't yet been compiled. Mix also remembers the exact version of each library it installs in the file mix.lock. This means that at any

point in the future you can get the same version of the library you use now.

Don’t worry that the library isn’t compiled—mix will automatically compile it the first time we need it.

If you look at your project tree, you’ll find a new directory called deps containing your dependencies. Note that these dependencies are themselves just projects, so you can browse their source and read their documentation.

[illegible]ser_agent)

Your Turn

- Exercise: OrganizingAProject-2

  Add the dependency to your project and install it.

Back to the Transformation

So, back to our problem. We have to write the function GithubIssues.fetch, which transforms a user name and project into a data structure containing that project's issues. The HTTPoison page on GitHub gives us a clue,[22] and we write a new module, Issues.GithubIssues:

project/1a/issues/lib/issues/github_issues.ex

We simply call get on the GitHub URL.

(We also have to pass in a user-agent header to keep the GitHub API happy.) What comes back is a structure. If we have a successful response, we return a tuple whose first element is :ok, along with the body. Otherwise we return an :error tuple, also with the body.

There's one more thing. The examples on the HTTPoison GitHub page call HTTPoison.start. That's because HTTPoison actually runs as a separate application, outside your main process. A lot of developers will copy this code, calling start inline like this.

In older versions of Elixir, you could add HTTPoison to the list of applications to start in mix.exs:

This is no longer necessary. The fact that you have listed HTTPoison as a dependency means that mix will automatically start it as an application.

What Does Application Mean?

OTP is the framework that manages suites of running applications. But just what *is* an application?

I found the answer counterintuitive at first. Erlang programs—and, by extension, Elixir programs—are often structured as suites of cooperating subapplications. Frequently, the code that would be a library in another language is a subapplication in Elixir. It might help to think of these as components or services.

We can play with this in IEx. Use the -S mix option to run mix before dropping into interaction mode. Because this is the first time we've tried to run our code since installing the dependencies, you'll see them get compiled:

Let’s try it out. (The output is massaged to fit the page.)

This tuple is the body of the GitHub response. The first element is set to :ok. The second element is a string containing the data encoded in JSON format.

Transformation: Convert Response

We'll need a JSON library to convert the response into a data structure. Searching hex.pm, I found the poison library (no relation to HTTPoison), so let's add its dependency to our mix.exs file.[23]

project/2/issues/mix.exs

Run mix deps.get, and you'll end up with poison installed.

To convert the body from a string, we call the Poison.Parser.parse! function when we return the message from the GitHub API:

project/3/issues/lib/issues/github_issues.ex

We also have to deal with a possible error response from the fetch, so back in the CLI module we write a function that decodes the body and returns it on a success response; the function extracts the error from the body and displays it otherwise.

project/3/issues/lib/issues/cli.ex

The JSON that GitHub returns for a successful response is a list of maps, where each map in the list contains a GitHub issue.

Dependencies That Aren't in Hex

The dependencies you need are likely to be in hex, so mix will probably find them automatically. However, sometimes you'll need to go further afield. The good

news is that mix can also load dependencies from other sources. The most common is GitHub.

HTTPoison uses a library called Hackney. In earlier versions of the book, Hackney wasn't in hex.pm, so I had to add the following dependency to my mix.exs:

Application Configuration

Before we move on, there's one little tweak I'd like to make. The issues_url function hard-codes the GitHub URL. Let's make this configurable.

Remember that when we created the project using mix new, it added a config/ directory containing config.exs. That file stores application-level configuration.

It should start with the line

We then write configuration information for each of the applications in our project. Here we're configuring the Issues application, so we write this code:

project/3a/issues/config/config.exs

Each config line adds one or more key/value pairs to the given application's environment. If you have multiple lines for the same application, they accumulate, with duplicate keys in later lines overriding values from earlier ones.

In our code, we use Application.get_env to return a value from the environment.

project/3a/issues/lib/issues/github_issues.ex

Because the application environment is commonly used in Erlang code, you'll find yourself using the configuration facility to configure code you import, as well as code you write.

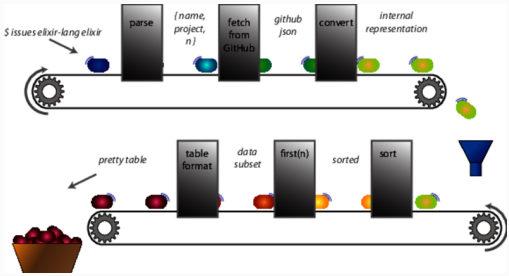
Sometimes you may want to vary the configuration, perhaps depending on your application's environment. One way is to use the import_config function, which reads configuration from a file. If your config.exs contains

then Elixir will read dev.exs, test.exs, or prod.exs, depending on your environment.

You can override the default config file name (config/config.exs) using the --config option to elixir.

Transformation: Sort Data

Have a look at the original design in the following figure.



We're making good progress—we've coded all the functions of the top conveyor belt. Our next transformation is to sort the data on its created_at field, with the newest entries first. And this can just use a standard Elixir library function, sort/2. We *could* create a new module for this, but it would be pretty lonely. For now we'll put the function in

the CLI module and keep an eye out for opportunities to move it out if we add related functions later.

So now our CLI module contains this:

project/3b/issues/lib/issues/cli.ex

That sort_into_descending_order function worries me a little—I get the comparison the wrong way around about 50% of the time, so let's write a little CLI test.

project/3b/issues/test/cli_test.exs

Update the import line at the top of the test:

and run it:

Lookin’ fine; mighty fine.

[illegible] Enum.reverse/1.

Transformation: Take First *n* Items

Our next transformation is to extract the first *count* entries from the list. Resisting the temptation to write the function ourselves (How *would* you write such a function?), we discover the built-in Enum.take:

Your Turn

- Exercise: OrganizingAProject-3

  Bring your version of this project in line with the code here.

- Exercise: OrganizingAProject-4

  (Tricky) Before reading the next section, see if you can write the code to format the data into columns, like the sample output at the start of the chapter. This is probably the longest piece of Elixir code you'll have written. Try to do it without using if or cond.

end

Transformation: Format the Table

All that's left from our design is to create the formatted table. The following would be a nice interface:

We pass the formatter the list of columns to include in the table, and it writes the table to standard output. The formatter doesn't add any new project- or design-related techniques, so we'll just show the listing.

project/4/issues/lib/issues/table_formatter.ex

And here are the tests for it:

project/4/issues/test/table_formatter_test.exs

(Although you can't see it here, the output we compare against in the last test contains trailing whitespace.)

Rather than clutter the process function in the CLI module with a long module name, I chose to use import to make the print function available without a module qualifier. This goes near the top of cli.ex.

This code also uses a great Elixir testing feature. By importing ExUnit.CaptureIO, we get access to the capture_io function. This runs the code passed to it but captures anything written to standard

output, returning it as a string.

Step 3: Make a Command-Line Executable

Although we can run our code by calling the run function via mix, it isn't friendly for other users. So let's create something we can run from the command line.

Mix can package our code, along with its dependencies, into a single file that can be run on any Unix-based platform. This uses Erlang's *escript* utility, which can run precompiled programs stored as a Zip archive. In our case, the program will be run as issues.

When escript runs a program, it looks in your mix.exs file for the option escript.

This should return a keyword list of escript configuration settings. The most important of these is main_module:, which must be set to the name of a module containing a main function. It passes the command-line arguments to this main function as a list of character lists (not binaries). As this seems to be a command-line concern, we'll put the main function in Issues.CLI. Here's the update to mix.exs:

project/4/issues/mix.exs

Now let's add a main function to our CLI. In fact, all we need to do is rename the existing run function:

project/4/issues/lib/issues/cli.ex

Then we package our program using mix:

Now we can run the app locally. We can also send it to a friend—it will run on any computer that has Erlang installed.

Step 4: Add Some Logging

Imagine a large Elixir application—dozens of processes potentially running across a number of nodes. You'd really want a standard way to keep track of significant events as it runs. Enter the Elixir logger.

The default mix.exs starts the logger for your application.

project/5/issues/mix.exs

The logger supports four levels of message—in increasing order of severity they are debug, info, warn, and error. You select the level of logging in

two ways.

First, you can determine at compile time the minimum level of logging to include. Logging below this level is not even compiled into your code. The compile-time level is set in the config/config.exs file:

project/5/issues/config/config.exs

Next, you can choose to change the minimum log level at runtime by calling Logger.configure. (Clearly, this cannot enable log levels that you excluded at compile time.)

After all this configuration, it's time to add some logging.

The basic logging functions are Logger.debug, .info, .warn, and .error. Each function takes either a string or a zero-arity function:

Why have the function version? Perhaps the calculation of the order total is expensive. In the first version, we'll always call it to interpolate the value into our string, even if the runtime log level is set to ignore debug-level messages. In the function variant, though, the total function will be invoked only if the log message is needed.

Anyway, here's a version of our fetch function with some logging:

project/5/issues/lib/issues/github_issues.ex

Note the use of require Logger at the top of the module. If you forget this (and I do every time), you'll get an error when you make the first call to Logger.

We can play with the new code in IEx:

Notice that the debug-level message is not displayed.

Step 5: Create Project Documentation

Java has Javadoc, Ruby has RDoc, and Elixir has ExDoc—a documentation tool that describes your project, showing the modules, the things defined in them, and any documentation you've written for them.

Using it is easy. First, add the ExDoc dependency to your mix.exs file. You'll also need to add an output formatter—I use earmark, a pure-Elixir Markdown-to-HTML convertor.

While you're in the mix.exs, you can add a project name and (if your project

is in GitHub) a URL. The latter allows ExDoc to provide live links to your source code. These parameters go in the project function:

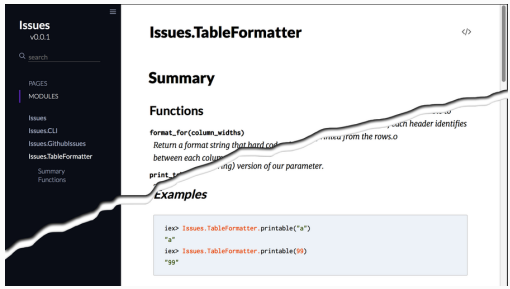
Then run mix deps.get.

To generate the documentation, just run

The first time you run this task, it will install ExDoc. That involves compiling some C code, so you'll need a development environment on your machine.

Open docs/index.html in your browser, then use the sidebar on the left to search

or drill down through your modules. Here's what I see for the start of the documentation for TableFormatter:

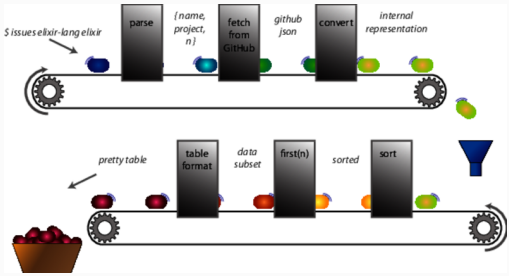


And that's it. The full project is in the source download at project/5/issues.

Coding by Transforming Data

I wanted to show you how Elixir projects are written—the tools we use and the processes we follow. I wanted to illustrate how lots of small functions can transform data, how specifying that transformation acts as an outline for the program, and how easy testing can be in Elixir.

But mostly I wanted to show how enjoyable Elixir development is, and how thinking about the world in terms of data and its transformation is a productive way to code. Look at our original design:



Then have a look at the CLI.process function:

This is a cool way to code. Next we'll dig into some of the tooling that makes using Elixir a joy.

Your Turn

- Exercise: OrganizingAProject-6

  In the United States, the National Oceanic and Atmospheric Administration provides hourly XML feeds of conditions at 1,800 locations.[24] For example, the feed for a small airport close to where I'm writing this is at http://w1.weather.gov/xml/current_obs/KDTO.xml.

  Write an application that fetches this data, parses it, and displays it in a nice format.

  (Hint: You might not have to download a library to handle XML parsing.)

Footnotes

[19] http://developer.github.com/v3/

[20] http://elixir-lang.org/getting-started/mix-otp/introduction-to-mix.html

[21] http://elixir-lang.org/docs/stable/elixir/OptionParser.html

[22] https://github.com/edgurgel/httpoison

[23]

https://github.com/devinus/poison

[24]

http://w1.weather.gov/xml/current_obs

Copyright © 2018, The Pragmatic Bookshelf.

We cover:

- *Debugging*
- *Testing*
- *Code exploration*
- *Server monitoring*
- *Source-code formatting*

Chapter 14

Tooling

You'd expect that a relatively new language would come with a fairly minimal set of tools—after all, the development team will be having fun playing with the language.

Not so with Elixir. Tooling was important from the start, and the core team has spent a lot of time providing a world-class environment in which to develop code.

In this short chapter, we'll look at some aspects of this.

This chapter is not the full list. We've already seen the ExDoc tool, which creates beautiful documentation for your code. Later, when we look at OTP applications,, we'll experiment with the Elixir release manager, a tool for managing releases while your application continues to run.

For now, let's look at testing, code-exploration, and server-monitoring

tools.

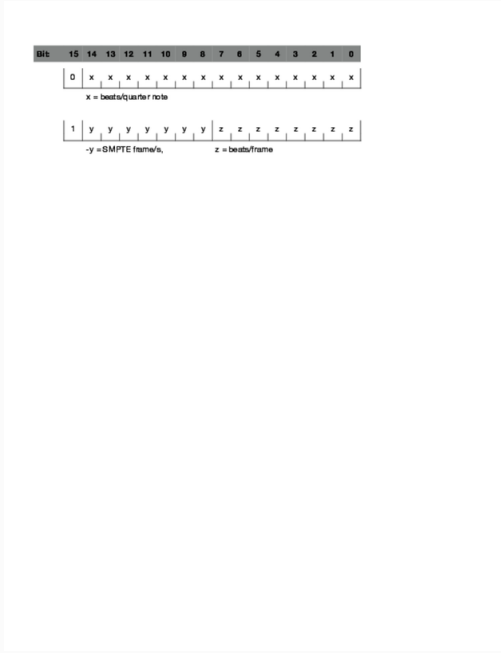
Debugging with IEx

You already know that IEx is the go-to utility to play with Elixir code. It also has a secret and dark second life as a debugger. It isn't fancy, but it lets you get into a running program and examine the environment.

You enter the debugger when running Elixir code hits a *breakpoint*. There are two ways of creating a breakpoint. One works by adding calls into the code you want to debug. The other is initiated from inside IEx. We'll look at both using the following (buggy) code:

tooling/buggy/lib/buggy.ex

This code is supposed to decode the data part of a MIDI header frame. This contains three 16-bit fields: the format, the number of tracks, and the time division. This last field comes in one of two formats:



The parse_header/1 function splits the overall header into the three fields, and

the decode/1 function works out which type of time division we have.

Let's run it, using a sample header I extracted from a MIDI file.

Oh no! That was totally unexpected. It looks like we're not passing the correct value to decode. Let's use the debugger to find out what's going on.

Injecting Breakpoints Using IEx.pry

We can add a breakpoint to our source code using the pry function. For example, to stop our code just before we call decode we could write this:

(We need the require because pry is a macro.)

Let's try the code now:

We reached the breakpoint, and IEx entered *pry mode*. It showed us the function we were in as well as the source lines surrounding the breakpoint.

At this point, IEx is running in the context of this function, so a call to binding shows the local variables. The value in the division function is 120, but that isn't matching either of the parameters to decode.

Aha! decode is expecting a binary, not an integer. Let's fix our code:

The pry call is still in there, so let's recompile and try again:

Now the division is a binary, and when we continue the code runs and outputs the header fields. Except…it's parsing the time division as if it were the SMPTE version, and not the beats/ quarter note version.

Setting Breakpoints with Break

The second way to create a breakpoint doesn't involve any code changes. Instead, you can use the break! command inside IEx to add a breakpoint on any public function. Let's remove the call to pry and run the code again. Inside

IEx we'll add a breakpoint on the decode function:

We hit the breakpoint, and we are indeed matching the wrong version of the decode function when we pass it 0000000001111000. Ah, that's because I'm discriminating based on the value of the top bit, and I got it the wrong way around: the SMPTE version should be

and the beats version should be

There's lots more functionality in the debugger. You can start by getting help for IEx.break/4.

Does This Seem a Little Artificial?

I have a confession to make. The only time I use the Elixir breakpoint facility is when I work on this section of the book. If I have to add code to the source to break in the middle of a function, then I can just raise an exception there instead to get the information I need. And the fact that I can only break at public functions from inside IEx means that I can't get the kind of granularity I need to diagnose issues, because 90% of my functions are private.

However, I'm an old curmudgeon—my favorite editor is a card punch. Don't let my lack of enthusiasm stop you from trying the debugger.

Testing

We already used the ExUnit framework to write tests for our Issues tracker app. But that chapter only scratched the surface of Elixir testing. Let's dig deeper.

Testing the Comments

When I document my functions, I like to include examples of the function being used—comments saying things such as, "Feed it these arguments, and you'll get this result." In the Elixir world, a common way to do this is to show the function being used in an IEx session.

Let's look at an example from our

Issues app. The TableFormatter formatter module defines a number of self-contained functions that we can document.

project/5/issues/lib/issues/table_formatter.ex

Note how some of the documentation contains sample IEx sessions. I like doing this. It helps people who come along later understand how to use my code. But, as importantly, it lets *me* understand what my code will feel like to use. I typically write these sample sessions before I start on the code, changing stuff around until the API feels right.

But the problem with comments is that they just don't get maintained. The code changes, the comment gets stale, and it becomes useless. Fortunately, ExUnit has doctest, a tool that extracts the iex sessions from your code's @doc strings, runs it, and checks that the output agrees with the comment.

To invoke it, simply add one or more

lines to your test files. You can add them to existing test files for a module (such as table_formatter_test.exs) or create a new test file just for them. That's what we'll do here. Let's create a new test file, test/doc_test.exs, containing this:

project/5/issues/test/doc_test.exs

We can now run it:

And, of course, these tests are integrated into the overall test suite:

Let’s force an error to see what happens:

And run the tests again:

Structuring Tests

You’ll often find yourself wanting to group your tests at a finer level than per

module. For example, you might have multiple tests for a particular function, or multiple functions that work on the same test data. ExUnit has you covered.

Let's test this simple module:

tooling/pbt/lib/stats.ex

Our tests might look something like this:

tooling/pbt/test/describe.exs

There are a couple of issues here. First, these tests only pass in a list of integers. Presumably we'd want to test with floats, too. So let's use the describe feature of ExUnit to document that these

are the integer versions of the tests:

tooling/pbt/test/describe.exs

If any of these fail, the message would include the description and test name:

A second issue with our tests is that we're duplicating the test data in each function. In this particular case this is arguably not a major problem. There are times, however, where this data is complicated to create. So let's use the setup feature to move this code into a single place. While we're at it, we'll also put the expected answers into the setup. This means that if we decide to change the test data in the future, we'll

find it all in one place.

tooling/pbt/test/describe.exs

The setup function is invoked automatically before each test is run. (There's also a setup_all function that is invoked just once for the test run.) The setup function returns a keyword list of named test data. In testing circles, this data, which is used to drive tests, is called a *fixture*.

This data is passed to our tests as a second parameter, following the test name. In my tests, I've called this parameter fixture. I then access the individual fields using the fixture.list

syntax.

In the code here I passed a block to setup. You can also pass the name of a function (as an atom).

Inside the setup code you can define callbacks using on_exit. These will be invoked at the end of the test. They can be used to undo changes made by the test.

There's a lot of depth in ExUnit. I'd recommend spending a little time in the ExUnit docs.[25]

Property-Based Testing

When you use assertions, you work out ahead of time the result you expect your

function to return. This is good, but it also has some limitations. In particular, any assumptions you made while writing the original code are likely to find their way into the tests, too.

A different approach is to consider the overall *properties* of the function you're testing. For example, if your function converts a string to uppercase, then you can predict that, whatever string you feed it,

- the length of the output will be the same as the length of the input, and
- any lowercase characters in the input will have been replaced by their uppercase counterpart.

These are intrinsic properties of the function. And we can test them statistically by simply injecting a (large)

number of different strings and verifying the results honor the properties. If all the tests pass, we haven't proved the function is correct (although we have a lot of confidence it should be). But, more importantly, if any of the tests fail, we've found a boundary condition our function doesn't handle. And property-based testing is surprisingly good at finding these errors. Let's look again at our previous example:

tooling/pbt/lib/stats.ex

Here are some simple properties we could test:

- The sum of a list containing a single value should be that value.
- The count function should never return a negative number.

- If we multiply the results returned by count and average, it should equal the result returned by sum (allowing for a little rounding).

To test the properties, the framework needs to generate large numbers of sample values of the correct type. For the first test, for example, we need a bunch of numeric values.

That's where the property-testing libraries come in. There are a number of property-based testing libraries for Elixir (including one I wrote, called Quixir). But here we'll be using a library called StreamData.[26] As José Valim is one of the authors, I suspect it may well make its way into core Elixir one day.

I could write something like this:

There are two pieces of magic here. The first is the real function. This is a generator, which will return real numbers. This is invoked by check all. You might think from its name that this will try all real numbers (which would take some time), but it instead just iterates a given number of times (100 by default).

Let's code this. First, add StreamData to our list of dependencies:

tooling/pbt/mix.exs

Now we can write the property tests. Here's the first:

tooling/pbt/test/stats_property_test.exs

You'll see this looks a lot like a regular test. We have to include `use ExCheck` at the top to include the property test framework.

The actual test is in the `property` block. It has the `check all` block we saw earlier. Inside this we have a test: `assert Stats.sum([number]) == number`.

Let's run it:

Let's break the test, just to see what a failure looks like:

We failed, and the value of number at the time was zero.

Let's fix the test, and add tests for the other two properties.

tooling/pbt/test/stats_property_test.exs

The two new tests use a different generator: list(int) generates a number of lists, each containing zero or more ints.

Running this code is surprising—it fails!

The exception shows we failed with an Arithmeticerror, and the value that caused the failure was `l = []`, the empty

list. That's because we were trying to find the average of an empty list. This means our code will be dividing the sum (0) by the list size (0), and dividing by zero is an error.

This is cool. The property tests explored the range of possible input values, and found one that causes our code to fail.

Arguably, this is a bug in our Stats module. But let's treat it instead as a boundary condition that the tests should avoid. We can do this in two ways.

First, we can tell the property test to skip generated values that fail to meet a condition. We do this with the nonempty function:

tooling/pbt/test/stats_property_test.exs

Now, whenever the generator returns an empty list, the nonempty function will filter it out. This is just one example of StreamData filters. A number are predefined, and you can also write your own.

A second approach is to prevent the generator from creating empty lists in the first place. This uses the min_length option:

tooling/pbt/test/stats_property_test.exs

Digging Deeper

In case you're interested in exploring property-based testing, the

documentation for ExUnitProperties has some examples and references.[27]

The StreamData module is designed to be used on its own—it's not just for testing. If you find yourself needing to generate streams of values that meet some criteria, it might be your library of choice.

Test Coverage

Some people believe that if there are any lines of application code that haven't been exercised by a test, the code is incomplete. (I'm not one of them.) These folks use *test coverage* tools to check for untested code.

Here we'll use excoveralls to see where to add tests for the Issues app.[28]

(Another good coverage tool is coverex.)[29]

All of the work to add the tool to our project takes place in mix.exs.

First, we add the dependency:

tooling/issues/mix.exs

Then, in the project section, we integrate the various coveralls commands into mix, and force them to run in the test environment:

tooling/issues/mix.exs

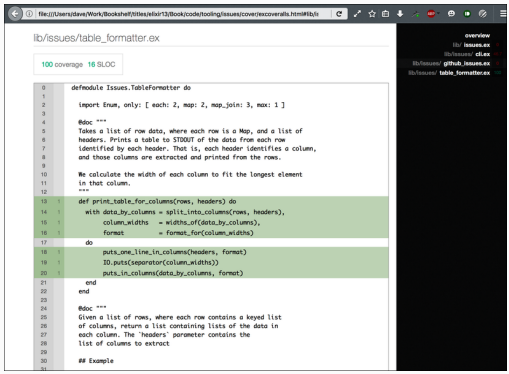
After a quick mix deps.get, you can run your first coverage report:

It runs the tests first, and then reports on the files in our application.

We have no tests for issues.ex. As this is basically a boilerplate no-op, that's not surprising. We wrote some tests for cli.ex, but could do better. The github_issues.ex file is not being tested. But, saving the best for last, we have 100% coverage in the table formatter (because we used it as an example of doc testing).

excoveralls can produce detailed reports to the console (mix coveralls.detail) and as an HTML file (mix coveralls.html). The latter generates the file cover/excoveralls.html, as shown in the

following figure.


lib/issues/table_formatter.ex
100 coverage 16 SLOC
overview
lib/ issues.ex
lib/issues/ cli.ex
lib/issues/ github_issues.ex
lib/issues/ table_formatter.ex
defmodule Issues.TableFormatter do
import Enum, only: [each: 2, map: 2, map_join: 3, max: 1]
@doc """
Takes a list of row data, where each row is a Map, and a list of
headers. Prints a table to STDOUT of the data from each row
identified by each header. That is, each header identifies a column,
and those columns are extracted and printed from the rows.
We calculate the width of each column to fit the longest element
in that column.
"""
def print_table_for_columns(rows, headers) do
with data_by_columns = split_into_columns(rows, headers),
column_widths = widths_of(data_by_columns),
format = format_for(column_widths)
do
puts_one_line_in_columns(headers, format)
IO.puts(separator(column_widths))
puts_in_columns(data_by_columns, format)
end
end
@doc """
Given a list of rows, where each row contains a keyed list
of columns, return a list containing lists of the data in
each column. The `headers` parameter contains the
list of columns to extract
Example

Finally, excoveralls works with a number of continuous integration systems. See its GitHub page for details.

Code Dependencies

The mix tool is smart when it compiles your project. It analyzes the dependencies between your source files, and only recompiles a file when it has changed or a file it depends on has changed. As a developer, you can also access this dependency information, gaining valuable insights into your code. You do this with the mix xref commands.

mix xref unreachable

List functions that are unknown at the time they are called.

mix xref warnings

List warnings associated with dependencies (for example, calls to unknown functions).

mix xref callers Mod | Mod.func | Mod.func/arity

List the callers to a module or function:

```
```

mix xref graph

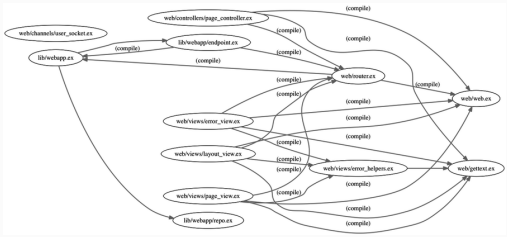
Show the dependency tree for the application:

```
```

You can produce a circles-and-arrows picture of dependencies using dot.[30]

```
```

This produces something like this:



Server Monitoring

As you might expect from a platform that has been running demanding and critical applications for 20 years, Erlang has great server-monitoring tools.

One of the easiest to use is already baked in. Inside IEx, run

```
```

Use this to get insight into…

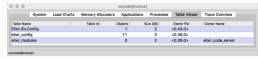
- Basic system information:



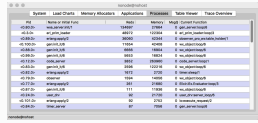
- Dynamic charts of load:



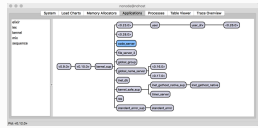
- Information and contents of Erlang ETS tables:



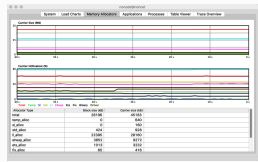
- Running processes:



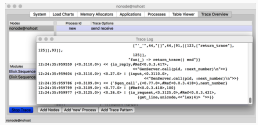
- Running applications:



- Memory allocation:



- And tracing of function calls, messages, and events:



For application-level monitoring, you might want to look at Elixometer from Pinterest.[31]

Source-Code Formatting

This is the section where I get into trouble.

The Elixir core team wanted to standardize the format of source code that was submitted to them for inclusion in the various Elixir core projects. Rather than beat people up and reject pull requests, they made it easy for submitters by including a source-code formatting tool in Elixir 1.6. This tool is pretty smart—it knows not just the syntax of Elixir but also the parse tree, meaning that it will often move things between lines, drop commas, add parentheses, and so on.

This magic is done using the mix format command. It can format single files, directory trees, and whole projects (see mix help format for information). This formatting replaces the files it touches, so you might want to make sure you're checked in before running it.

Let's have a look at some before-and-after formatting:

If we feed it code that looks like a dog's dinner:

the formatter tidies it nicely:

It is also pretty smart about multiline

constructs:

This produces:

Here, because the original split onto a new line, the formatted result was normalized into one element per line.

There's lots to like about the formatter. But I personally don't use it, because it destroys some elements of layout I think are important.

For example, I like to line things up vertically. I find this much, much easier to read and to maintain. Most editors have support for this:

and:

Unfortunately, the Elixir formatter goes to some trouble to remove that extra space, producing:

Then there's the contentious trailing comma.

When I list things out, I put a comma after each item and put each item on a new line. This makes it easier to move things around, add new items, sort the list, and so on. Each line is just like the other: the first and last lines are not special.

The formatter thinks this is silly, and removes the comma.

And finally, there's the trailing comment. I rarely use comments inside a block of code. When I do, and if it is short, I add it to the end of the line:

I know this is a bad example, but even so the formatter makes some unfortunate decisions:

The upshot? If you like what it does, or if you're submitting code to a project that requires it, use the formatter.

Inevitably, There's More

Elixir is lucky when it comes to tooling, both because it inherits a wealth of tools from Erlang and because the Elixir community values great tools and develops them to fill any gaps. Keep up with the tools people use, and you'll find yourself developing faster, and with more confidence.

Now let's look at concurrent programming, a key strength of Elixir.

Footnotes

[25] http://elixir-lang.org/docs/stable/ex_unit/ExUnit.html

[26] https://github.com/whatyouhide/stream_data

[27]

https://hexdocs.pm/stream_data/ExUnitProperties.html

[28]

https://github.com/parroty/excoveralls

[29]

https://github.com/alfert/coverex

[30]

http://www.graphviz.org/

[31]

https://github.com/pinterest/elixometer

Copyright © 2018, The Pragmatic Bookshelf.

Part 2
Concurrent Programming

You want to write concurrent programs. That's probably why you're reading this book.

Let's look at Elixir's actor-based concurrency model. Then we'll dig into OTP, the Erlang management architecture that helps you create applications that are highly scalable and very reliable.


Comte Anthony Eden
@aeden
Follow
All your objects want to grow up to be actors.
Reply Retweet Favorite More
1:16 AM - 16 Jun 13 from Limoux, Aude
Chad Fowler @chadfowler
21h
@aeden sadly, most just end up waiting tables
Details Reply Retweet Favorite More

Chapter 15

Working with Multiple Processes

One of Elixir's key features is the idea of packaging code into small chunks that can be run independently and concurrently.

If you've come from a conventional programming language, this may worry you. Concurrent programming is "known" to be difficult, and there's a performance penalty to pay when you create lots of processes.

Elixir doesn't have these issues, thanks

to the architecture of the Erlang VM on which it runs.

Elixir uses the *actor* model of concurrency. An actor is an independent process that shares nothing with any other process. You can spawn new processes, send them messages, and receive messages back. And that's it (apart from some details about error handling and monitoring, which we cover later).

In the past, you may have had to use threads or operating system processes to achieve concurrency. Each time, you probably felt you were opening Pandora's box—there was so much that could go wrong. But that worry just evaporates in Elixir. In fact, Elixir

developers are so comfortable creating new processes, they'll often do it at times when you'd have created an *object* in a language such as Java.

One more thing—when we talk about processes in Elixir, we are not talking about native operating-system processes. These are too slow and bulky. Instead, Elixir uses process support in Erlang. These processes will run across all your CPUs (just like native processes), but they have very little overhead. As we'll cover a bit later, it's very easy to create hundreds of thousands of Elixir processes on even a modest computer.

[illegible]ic,

A Simple Process

Here's a module that defines a function we'd like to run as a separate process:

spawn/spawn-basic.ex

Yup, that's it. There's nothing special—it's just regular code.

Let's fire up IEx and play:

First let's call it as a regular function:

Now let's run it in a separate process:

The spawn function kicks off a new process. It comes in many forms, but the two simplest ones let you run an anonymous function and run a named function in a module, passing a list of arguments. (We used the latter here.)

The spawn returns a *process identifier*, normally called a PID. This uniquely identifies the process it creates. (This identifier could be unique among all processes in the world, but here it's just unique in our application.)

When we call spawn, it creates a new process to run the code we specify. We don't know exactly when it will execute —only that it is eligible to run.

In this example, we can see that our

function ran and output “Hello” prior to IEx reporting the PID returned by spawn. But you can’t rely on this. Instead you’ll use messages to synchronize your processes’ activity.

Sending Messages Between Processes

Let’s rewrite our example to use messages. The top level will send greet a message containing a string, and the greet function will respond with a greeting containing that message.

In Elixir we send a message using the send function. It takes a PID and the message to send (an Elixir value, which we also call a *term*) on the right. You can send anything you want, but most Elixir developers seem to use atoms and tuples.

We wait for messages using receive. In a way, this acts the same as case, with the message body as the parameter. Inside the block associated with the receive call, you can specify any number of patterns and associated actions. Just as with case, the action associated with the first pattern that matches the function is run.

Here's the updated version of our greet function.

spawn/spawn1.exs

The function uses receive to wait for a message, and then matches the message in the block. In this case, the only pattern is a two-element tuple, where the

first element is the original sender's PID and the second is the message. In the corresponding action, we use send sender, ... to send a formatted string back to the original message sender. We package that string into a tuple, with :ok as its first element.

Outside the module, we call spawn to create a process, and send it a tuple:

The function self returns its caller's PID. Here we use it to pass our PID to the greet function so it will know where to send the response.

We then wait for a response. Notice that we do a pattern match on {:ok,

message}, extracting the second element of the tuple, which contains the actual text.

We can run this in IEx:

Very cool. The text was sent, and greet responded with the full greeting.

Handling Multiple Messages

Let's try sending a second message.

spawn/spawn2.exs

Run it in IEx:

The first message is sent back, but the

second is nowhere to be seen. What's worse, IEx just hangs, and we have to use ^C (the Control-C key sequence) to get out of it.

That's because our greet function handles only a single message. Once it has processed the receive, it exits. As a result, the second message we send it is never processed. The second receive at the top level then just hangs, waiting for a response that will never come.

Let's at least fix the hanging part. We can tell receive that we want to time out if a response is not received in so many milliseconds. This uses a pseudopattern called after.

spawn/spawn3.exs

But how would we make our greet function handle multiple messages? Our natural reaction is to make it loop, doing a receive on each iteration. Elixir doesn't have loops, but it does have recursion.

spawn/spawn4.exs

Run this, and both messages are processed:

Recursion, Looping, and the Stack

The greet function might have worried you a little. Every time it receives a message, it ends up calling itself. In

many languages, that adds a new frame to the stack. After a large number of messages, you might run out of memory.

This doesn't happen in Elixir, as it implements *tail-call optimization*. If the last thing a function does is call itself, there's no need to make the call. Instead, the runtime simply jumps back to the start of the function. If the recursive call has arguments, then these replace the original parameters. But beware—the recursive call *must* be the very last thing executed. For example, the following code is not tail recursive:

While the recursive call is physically the

last thing in the function, it is not the last thing executed. The function has to multiply the value it returns by *n*.

To make it tail recursive, we need to move the multiplication into the recursive call, and this means adding an accumulator:

spawn/fact_tr.exs

end(next_pid)
end

send(next_pid, n+1)
end
end

Process Overhead

At the start of the chapter, I somewhat cavalierly said Elixir processes were very low overhead. Now it's time to back that up. Let's write some code that creates *n* processes. The first will send a number to the second. It will increment that number and pass it to the third. This will continue until we get to the last process, which will pass the number back to the top level.

spawn/chain.exs

The counter function on line 2 is the code that will be run in separate processes. It is passed the PID of the

next process in the chain. When it receives a number, it increments it and sends it on to that next process.

The create_processes function is probably the densest piece of Elixir we've encountered so far. Let's break it down.

It is passed the number of processes to create. Each process has to be passed the PID of the previous process so that it knows who to send the updated number to.

The code that creates each process is defined in a one-line anonymous function, which is assigned to the variable code_to_run. The function takes two parameters because we're passing it

to Enum.reduce on line 14.

The reduce call will iterate over the range 1..n. Each time around, it will pass an accumulator as the second parameter to its function. We set the initial value of that accumulator to self, our PID.

In the function, we spawn a new process that runs the counter function, using the third parameter of spawn to pass in the accumulator's current value (initially self). The value spawn returns is the PID of the newly created process, which becomes the accumulator's value for the next iteration.

Putting it another way, each time we spawn a new process, we pass it the

previous process's PID in the send_to parameter.

The value that the reduce function returns is the accumulator's final value, which is the PID of the last process created.

On the next line we set the ball rolling by passing 0 to the last process. The process increments the value and so passes 1 to the second-to-last process. This goes on until the very first process we created passes the result back to us. We use the receive block to capture this, and format the result into a nice message.

Our receive block contains a new feature. We've already seen how guard

clauses can constrain pattern matching and function calling. The same guard clauses can be used to qualify the pattern in a receive block.

Why do we need this, though? It turns out there's a bug in some versions of Elixir.[32] When you compile and run a program using iex -S mix, a residual message is left lying around from the compilation process (it records a process's termination). We ignore that message by telling the receive clause that we're interested only in simple integers.

The run function starts the whole thing off. It uses a built-in Erlang library, tc, which can time a function's execution. We pass it the module, name, and

parameters, and it responds with a tuple. The first element is the execution time in microseconds and the second is the result the function returns.

We'll run this code from the command line rather than from IEx. (You'll see why in a second.) These results are on my 2011 MacBook Air (2.13 GHz Core 2 Duo and 4 GB of RAM).

We asked it to run 10 processes, and it came back in 4 ms. The answer looks correct. Let's try 100 processes.

Only a small increase in the time. There's probably some startup latency

on the first process creation. Onward! Let's try 1,000.

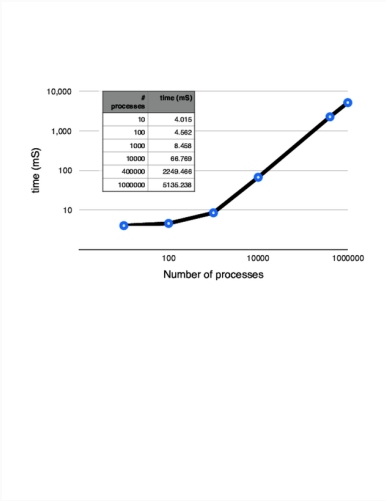
Now 10,000.

Ten thousand processes created and executed in 66 ms. Let's try for 400,000.

It looks like the virtual machine won't support 400,000 processes. Fortunately, this is not a hard limit—we just bumped into a default value. We can increase this using the VM's `+P` parameter. We pass this parameter to the VM using the `--erl` parameter to `elixir`. (This is why I chose to run from the command line.)

One last run, this time with 1,000,000 processes.

We ran a million processes (sequentially) in just over 5 seconds. And, as the graph shows, the time per process was pretty much linear once we overcame the startup time.



# processes	time (mS)
10	4.015
100	4.562
1000	8.458
10000	66.769
400000	2249.466
1000000	5135.238

This kind of performance is stunning,

and it changes the way we design code. We can now create hundreds of little helper processes. And each process can contain its own state—in a way, processes in Elixir are like objects in an object-oriented system (but they're more self-contained).

Your Turn

- Exercise: WorkingWithMultipleProcesses-1

  Run this code on your machine. See if you get comparable results.

- Exercise: WorkingWithMultipleProcesses-2

  Write a program that spawns two processes and then passes each a unique token (for example, “fred” and “betty”). Have them send the tokens back.

    - Is the order in which the replies are received deterministic in theory? In practice?

    - If either answer is no, how could you make it so?

When Processes Die

Who gets told when a process dies? By default, no one. Obviously the VM knows and can report it to the console, but your code will be oblivious unless you explicitly tell Elixir you want to get involved. Here's the default case: we spawn a function that uses the Erlang timer library to sleep for 500 ms. It then exits with a status of :boom. The code that spawns it sits in a receive. If it receives a message, it reports that fact; otherwise, after one second it lets us know that nothing happened.

spawn/link1.exs

(Think about how you'd have written this in your old programming language.)

We can run this from the console:

The top level got no notification when the spawned process exited.

Linking Two Processes

If we want two processes to share in each other's pain, we can *link* them. When processes are linked, each can receive information when the other exits. The spawn_link call spawns a process and links it to the caller in one operation.

spawn/link2.exs

The runtime reports the abnormal termination:

So our child process died, and it killed the entire application. That's the default behavior of linked processes—when one exits abnormally, it kills the other.

What if you want to handle the death of another process? Well, you probably don't want to do this. Elixir uses the OTP framework for constructing process trees, and OTP includes the concept of process supervision. An incredible amount of effort has been spent getting this right, so I recommend using it most of the time. (We cover this in Chapter 18, *OTP: Supervisors*.)

However, you can tell Elixir to convert the exit signals from a linked process into a message you can handle. Do this by *trapping the exit*.

spawn/link3.exs

This time we see an :EXIT message when the spawned process terminates:

It doesn't matter why a process exits—it may simply finish processing, it may explicitly exit, or it may raise an exception—the same :EXIT message is received. Following an error, however, it contains details of what went wrong.

Monitoring a Process

Linking joins the calling process and another process—each receives notifications about the other. By contrast, *monitoring* lets a process spawn another and be notified of its termination, but without the reverse notification—it is one-way only.

When you monitor a process, you receive a :DOWN message when it exits or fails, or if it doesn't exist.

You can use spawn_monitor to turn on monitoring when you spawn a process, or you can use Process.monitor to monitor an existing process. However, if you use Process.monitor (or link to an existing process), there is a potential race condition—if the other process dies before your monitor call completes, you

may not receive a notification. The spawn_link and spawn_monitor versions are atomic, however, so you'll always catch a failure.

spawn/monitor1.exs

Run it, and the results are similar to the spawn_link version:

(The Reference record in the message is the identity of the monitor that was created. The spawn_monitor call also returns it, along with the PID.)

So, when do you use links and when should you choose monitors?

It depends on your processes' semantics. If the intent is that a failure in one process should terminate another, then you need links. If instead you need to know when some other process exits for any reason, choose monitors.

Your Turn

The Erlang function timer.sleep(time_in_ms) suspends the current process for a given time. You might want to use it to force some scenarios in the following exercises. The key with the exercises is to get used to the different reports you'll see when you're developing code.

- Exercise: WorkingWithMultipleProcesses-3

  Use spawn_link to start a process, and have that process send a message to the parent and then exit immediately. Meanwhile, sleep for 500 ms in the parent, then receive as many messages as are waiting. Trace what you receive. Does it matter that you weren't waiting for the notification from the child when it exited?

- Exercise: WorkingWithMultipleProcesses-4

Do the same, but have the child raise an exception. What difference do you see in the tracing?

- Exercise: WorkingWithMultipleProcesses-5

  Repeat the two, changing spawn_link to spawn_monitor.

ction,

Parallel Map—The “Hello, World” of Erlang

Devin Torres reminded me that every book in the Erlang space must, by law, include a parallel map function. Regular map returns the list that results from applying a function to each element of a collection. The parallel version does the same, but it applies the function to each element in a separate process.

spawn/pmap.exs

Our method contains two transformations (look for the |> operator). The first transformation maps collection into a list of PIDs, where each

PID in the list runs the given function on an individual list element. If the collection contains 1,000 items, we'll run 1,000 processes.

The second transformation converts the list of PIDs into the results returned by the processes corresponding to each PID in the list. Note how it uses ^pid in the receive block to get the result for each PID in turn. Without this we'd get back the results in random order.

But does it work?

That's pretty sweet, but it gets better, as we'll cover when we look at tasks and agents.

Your Turn

- Exercise: WorkingWithMultipleProcesses-6

  In the pmap code, I assigned the value of self to the variable me at the top of the method and then used me as the target of the message returned by the spawned processes. Why use a separate variable here?

- Exercise: WorkingWithMultipleProcesses-7

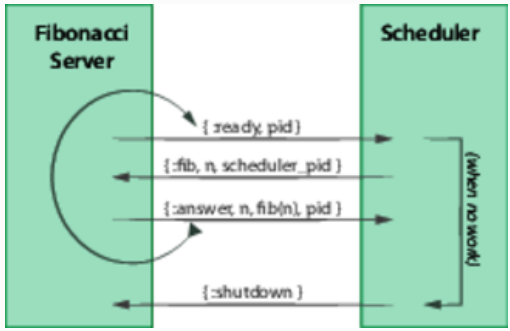
  Change the ^pid in pmap to _pid. This means the receive block will take responses in the order the processes send them. Now run the code again. Do you see any difference in the output? If you're like me, you don't, but the program clearly contains a bug. Are you scared by this? Can you find a way to reveal the problem (perhaps by passing in a different function, by sleeping, or by increasing the number of processes)? Change it back to ^pid and make sure the order is now correct.

A Fibonacci Server

Let's round out this chapter with an example program. Its task is to calculate *fib(n)* for a list of *n*, where *fib(n)* is the nth Fibonacci number. (The Fibonacci sequence starts 0, 1. Each subsequent number is the sum of the preceding two numbers in the sequence.)[33] I chose this not because it is something we all do every day, but because the naive calculation of Fibonacci numbers 10 through 37 takes a measurable number of seconds on typical computers.

The twist is that we'll write our program to calculate different Fibonacci numbers in parallel. To do this, we'll write a

trivial server process that does the calculation, and a scheduler that assigns work to a calculation process when it becomes free. The following diagram shows the message flow.



When the calculator is ready for the next number, it sends a :ready message to the scheduler. If there is still work to do, the scheduler sends it to the calculator in a :fib message; otherwise it sends the calculator a :shutdown. When a calculator receives a :fib message, it calculates the given Fibonacci number

and returns it in an :answer. If it gets a :shutdown, it simply exits.

Here's the Fibonacci calculator module:

spawn/fib.exs

The public API is the fib function, which takes the scheduler PID. When it starts, it sends a :ready message to the scheduler and waits for a message back.

If it gets a :fib message, it calculates the answer and sends it back to the client. It then loops by calling itself recursively. This will send another :ready message, telling the client it is ready for more work.

If it gets a :shutdown it simply exits.

The Task Scheduler

The scheduler is a little more complex, as it is designed to handle both a varying number of server processes and an unknown amount of work.

spawn/fib.exs

The public API for the scheduler is the run function. It receives the number of processes to spawn, the module and function to spawn, and a list of things to process. The scheduler is pleasantly ignorant of the actual task being performed.

Let's emphasize that last point. Our

scheduler knows nothing about Fibonacci numbers. Exactly the same code will happily manage processes working on DNA sequencing or password cracking.

The run function spawns the correct number of processes and records their PIDs. It then calls the workhorse function, schedule_processes.

This function is basically a receive loop. If it gets a :ready message from a server, it sees if there is more work in the queue. If there is, it passes the next number to the calculator and then recurses with one fewer number in the queue.

If the work queue is empty when it

receives a :ready message, it sends a shutdown to the server. If this is the last process, then we're done and it sorts the accumulated results. If it isn't the last process, it removes the process from the list of processes and recurses to handle another message.

Finally, if it gets an :answer message, it records the answer in the result accumulator and recurses to handle the next message.

We drive the scheduler with the following code:

spawn/fib.exs

The to_process list contains the numbers

we'll be passing to our fib servers. In our case, we give it the same number, 37, 20 times. The intent here is to load each of our processors.

We run the code a total of 10 times, varying the number of spawned processes from 1 to 10. We use :timer.tc to determine the elapsed time of each iteration, reporting the result in seconds. The first time around the loop, we also display the numbers we calculated.

Cody Russell kindly ran this for me on his four-core system. He saw a dramatic reduction in elapsed time when we increase the concurrency from one to two, small decreases until we hit four

processes, then fairly flat performance after that. The Activity Monitor showed a consistent 380% CPU use once the concurrency got above 4. (If you want to see similar results on systems with more cores, you'll need to increase the number of entries in the to_process list.)

Your Turn

- Exercise: WorkingWithMultipleProcesses-8

  Run the Fibonacci code on your machine. Do you get comparable timings? If your machine has multiple cores and/or processors, do you see improvements in the timing as we increase the application's concurrency?

- Exercise: WorkingWithMultipleProcesses-9

  Take this scheduler code and update it to let you run a function that finds the number of times the word "cat" appears in each file in a given directory. Run one server process per file. The function `File.ls!` returns the names of files in a directory, and `File.read!` reads the contents of a file as a binary. Can you write it as a more generalized scheduler?

  Run your code on a directory with a reasonable number of files (maybe around 100) so you can experiment with the effects of concurrency.

fib(0)

Agents—A Teaser

Our Fibonacci code is really inefficient. To calculate fib(5), we calculate this:

Look at all that duplication. If only we could cache the intermediate values.

As you know, Elixir modules are basically buckets of functions—they cannot hold state. But processes can hold state. And Elixir comes with a library module called Agent that makes it easy to wrap a process containing state in a nice module interface. Don't worry about the details of the code that

follows—we cover agents and tasks. For now, just see how processes are among the tools we use to add persistence to Elixir code. (This code comes from a mailing-list post by José Valim, written in response to some ugly code I wrote.)

spawn/fib_agent.exs

Let's run it:

If we'd tried to calculate fib(2000) using the noncached version, the sun would grow to engulf the Earth while we were waiting for it to finish.

Thinking in Processes

If you first started programming with procedural languages and then moved to an object-oriented style, you'll have experienced a period of dislocation as you tried to get your head to think in terms of objects.

The same will be happening now as you start to think of your work in terms of processes. Just about every decent Elixir program will have many, many processes, and by and large they'll be just as easy to create and manage as the objects were in object-oriented programming. But learning to think that way takes awhile. Stick with it.

So far we've been running our processes in the same VM. But if we're planning on taking over the world, we need to be able to scale. And that means running on more than one machine.

The abstraction for this is the *node*, and that's the subject of the next chapter.

Footnotes

[32] https://github.com/elixir-lang/elixir/issues/1050

[33] http://en.wikipedia.org/wiki/Fibonacci_number

Copyright © 2018, The Pragmatic Bookshelf.

Chapter 16

Nodes—The Key to Distributing Services

There's nothing mysterious about a *node*. It is simply a running Erlang VM. Throughout this book we've been running our code on a node.

The Erlang VM, called *Beam*, is more than a simple interpreter. It's like its own little operating system running on top of your host operating system. It handles its own events, process scheduling, memory, naming services, and interprocess communication. In addition to all that, a node can connect

to other nodes—in the same computer, across a LAN, or across the Internet—and provide many of the same services across these connections that it provides to the processes it hosts locally.

Naming Nodes

So far we haven't needed to give our node a name—we've had only one. If we ask Elixir what the current node is called, it'll give us a made-up name:

We can set the name of a node when we start it. With IEx, use either the --name or --sname option. The former sets a fully qualified name:

The latter sets a short name:

The name that's returned is an atom—

it's in quotes because it contains characters not allowed in a literal atom.

Note that in both cases the IEx prompt contains the node's name along with my machine's name (light-boy).

Now I want to show you what happens when we have two nodes running. The easiest way to do this is to open two terminal windows and run a node in each. To represent these windows in the book, I'll show them stacked vertically.

Let's run a node called node_one in the top window and node_two in the bottom one. We'll then use the Elixir Node module's list function to display a list of known nodes, then connect from one to

the other.

Initially, node_two doesn't know about any other nodes. But after we connect to node_one (notice that we pass an atom containing that node's name), the list shows the other node. And if we go back to node one, it will now know about node two.

Now that we have two nodes, we can try running some code. On node one, let's create an anonymous function that outputs the current node name.

We can run this with the spawn

function.

But spawn also lets us specify a node name. The process will be spawned on that node.

We're running on node one. When we tell spawn to run on node_one@light-boy, we see two lines of output. The first is the PID spawn returns, and the second is the value of Node.self that the function writes.

The second spawn is where it gets interesting. We pass it the name of node two and the same function we used the first time. Again we get two lines of

output. The first is the PID and the second is the node name. Notice the PID's contents. The first field in a PID is the node number. When running on a local node, it's zero. But here we're running on a remote node, so that field has a positive value (7393). Then look at the function's output. It reports that it is running on node two. I think that's pretty cool.

You may have been expecting the output from the second spawn to appear in the lower window. After all, the code runs on node two. But it was created on node one, so it inherits its process hierarchy from node one. Part of that hierarchy is something called the *group leader*, which (among other things)

determines where IO.puts sends its output. So in a way, what we're seeing is doubly impressive. We start on node one, run a process on node two, and when the process outputs something, it appears back on node one.

puts(Enum.join(File.ls!,

Your Turn

- Exercise: Nodes-1

  Set up two terminal windows, and go to a different directory in each. Then start up a named node in each. In one window, write a function that lists the contents of the current directory.

  Run it twice, once on each node.

Nodes, Cookies, and Security

Although this is cool, it might also ring some alarm bells. If you can run arbitrary code on any node, then anyone with a publicly accessible node has just handed over his machine to any random hacker.

But that's not the case. Before a node

will let another connect, it checks that the remote node has permission. It does that by comparing that node's *cookie* with its own cookie. A cookie is just an arbitrary string (ideally fairly long and very random). As an administrator of a distributed Elixir system, you need to create a cookie and then make sure all nodes use it.

If you are running the iex or elixir commands, you can pass in the cookie using the --cookie option.

If we repeat our two-node experiment and explicitly set the cookie names to be different, what happens?

The node that attempts to connect receives false, indicating the connection was not made. And the node that it tried to connect to logs an error describing the attempt.

But why does it succeed when we don't specify a cookie? When Erlang starts, it looks for an .erlang.cookie file in your home directory. If that file doesn't exist, Erlang creates it and stores a random string in it. It uses that string as the cookie for any node the user starts. That way, all nodes you start on a particular machine are automatically given access to each other.

Be careful when connecting nodes over a public network—the cookie is transmitted in plain text.

Naming Your Processes

Although a PID is displayed as three numbers, it contains just two fields; the first number is the node ID and the next two numbers are the low and high bits of the process ID. When you run a process on your current node, its node ID will always be zero. However, when you export a PID to another node, the node ID is set to the number of the node on which the process lives.

That works well once a system is up and running and everything is knitted together. If you want to register a callback process on one node and an event-generating process on another,

just give the callback PID to the generator.

But how can the callback find the generator in the first place? One way is for the generator to register its PID, giving it a name. The callback on the other node can look up the generator by name, using the PID that comes back to send messages to it.

Here's an example. Let's write a simple server that sends a notification about every 2 seconds. To receive the notification, a client has to register with the server. And we'll arrange things so that clients on different nodes can register.

While we're at it, we'll do a little

packaging so that to start the server you run Ticker.start, and to start the client you run Client.start. We'll also add an API Ticker.register to register a client with the server.

Here's the server code:

nodes/ticker.ex

We define a start function that spawns the server process. It then uses :global.register_name to register the PID of this server under the name :ticker.

Clients who want to register to receive ticks call the register function. This function sends a message to the Ticker

server, asking it to add those clients to its list. Clients could have done this directly by sending the :register message to the server process. Instead, we give them an interface function that hides the registration details. This helps decouple the client from the server and gives us more flexibility to change things in the future.

Before we look at the actual tick process, let's stop to consider the start and register functions. These are not part of the tick process—they are simply chunks of code in the Ticker module. This means they can be called directly wherever we have the module loaded—no message passing required. This is a common pattern; we have a module that

is responsible both for spawning a process and for providing the external interface to that process.

Back to the code. The last function, generator, is the spawned process. It waits for two events. When it gets a tuple containing :register and a PID, it adds the PID to the list of clients and recurses. Alternatively, it may time out after 2 seconds, in which case it sends a {:tick} message to all registered clients.

(This code has no error handling and no means of terminating the process. I just wanted to illustrate passing PIDs and messages between nodes.) The client code is simple:

nodes/ticker.ex

It spawns a receiver to handle the incoming ticks, and passes the receiver's PID to the server as an argument to the register function. Again, it's worth noting that this function call is local—it runs on the same node as the client. However, inside the Ticker.register function, it locates the node containing the server and sends it a message. As our client's PID is sent to the server, it becomes an external PID, pointing back to the client's node.

The spawned client process simply loops, writing a cheery message to the console whenever it receives a tick message.

Let's run it. We'll start up our two

nodes. We'll call Ticker.start on node one. Then we'll call Client.start on both node one and node two.

To stop this, you'll need to exit IEx on both nodes.

When to Name Processes

When you name something, you are recording some global state. And as we all know, global state can be troublesome. What if two processes try to register the same name, for example?

The runtime has some tricks to help us. In particular, we can list the names our application will register in the app's mix.exs file. (We'll cover how when we look at packaging an application.)

However, the general rule is to register your process names when your application starts.

Your Turn

- Exercise: Nodes-2

  When I introduced the interval server, I said it sent a tick “about every 2 seconds.” But in the receive loop, it has an explicit timeout of 2,000 ms. Why did I say “about” when it looks as if the time should be pretty accurate?

- Exercise: Nodes-3

  Alter the code so that successive ticks are sent to each registered client (so the first goes to the first client, the second to the next client, and so on). Once the last client receives a tick, the process starts back at the first. The solution should deal with new clients being added at any time.

Input, Output, PIDs, and Nodes

Input and output in the Erlang VM are performed using I/O servers. These are simply Erlang processes that implement a low-level message interface. You never have to deal with this interface directly (which is a good thing, as it is complex). Instead, you use the various Elixir and Erlang I/O libraries and let them do the heavy lifting.

In Elixir you identify an open file or device by the PID of its I/O server. And these PIDs behave like all other PIDs—you can, for example, send them between nodes. If you look at the implementation of Elixir's IO.puts

function, you'll see

(To see the source of an Elixir library module, view the online documentation at http://elixir-lang.org/docs/, navigate to the function in question, and click the *Source* link.)

The default device it uses is returned by the function :erlang.group_leader. (The group_leader function is imported from the :erlang module at the top of the IO module.) This will be the PID of an I/O server.

So, bring up two terminal windows and start a different named node in each. Connect to node one from node two, and

register the PID returned by group_leader under a global name (we use :two).

Note that once we've registered the PID, we can access it from the other node. And once we've done that, we can pass it to IO.puts; the output appears in the other terminal window.

Your Turn

- Exercise: Nodes-4

  The ticker process in this chapter is a central server that sends events to registered clients. Reimplement this as a ring of clients. A client sends a tick to the next client in the ring. After 2 seconds, *that* client sends a tick to *its* next client.

  When thinking about how to add clients to the ring, remember to deal with the case where a client's receive loop times out just as you're adding a new process. What does this say about who has to be responsible for updating the links?

Nodes Are the Basis of Distribution

We've seen how we can create and interlink a number of Erlang virtual machines, potentially communicating across a network. This is important, both to allow your application to scale and to increase reliability. Running all your code on one machine is like having all your eggs in one basket. Unless you're writing a mobile omelet app, this is probably not a good idea.

It's easy to write concurrent applications with Elixir. But writing code that follows the happy path is a lot easier than writing bullet-proof, scalable, and hot-swappable world-beating apps. For

that, you’re going to need some help.

In the worlds of Elixir and Erlang, that help is called OTP, and it is the subject of the next few chapters.

Copyright © 2018, The Pragmatic Bookshelf.

Chapter 17

OTP: Servers

If you've been following Elixir or Erlang, you've probably come across OTP. It is often hyped as the answer to all high-availability distributed-application woes. It isn't, but it certainly solves many problems that you'd otherwise need to solve yourself, including application discovery, failure detection and management, hot code swapping, and server structure.

First, the obligatory one-paragraph history. OTP stands for the *Open Telecom Platform*, but the full name is

largely of historical interest and everyone just says *OTP*. It was initially used to build telephone exchanges and switches. But these devices have the same characteristics we want from any large online application, so OTP is now a general-purpose tool for developing and managing large systems.

OTP is actually a bundle that includes Erlang, a database (wonderfully called *Mnesia*), and an innumerable number of libraries. It also defines a structure for your applications. But, as with all large, complex frameworks, there is a lot to learn. In this book we'll focus on the essentials and I'll point you toward other information sources.

We've been using OTP all along—mix,

the Elixir compiler, and even our issue tracker followed OTP conventions. But that use was implicit. Now we'll make it explicit and start writing servers using OTP.

Some OTP Definitions

OTP defines systems in terms of hierarchies of *applications*. An application consists of one or more processes. These processes follow one of a small number of OTP conventions, called *behaviors*. There is a behavior used for general-purpose servers, one for implementing event handlers, and one for finite-state machines. Each implementation of one of these behaviors will run in its own process (and may have additional associated processes). In this chapter we'll be implementing the *server* behavior, called *GenServer*.

A special behavior, called *supervisor*, monitors the health of these processes and implements strategies for restarting them if needed.

We'll take a look at these components from the bottom up—this chapter will cover servers, the next will explore supervisors, and finally we'll implement applications.

An OTP Server

When we wrote our Fibonacci server in the previous chapter,, we had to do all the message handling ourselves. It wasn't difficult, but it was tedious. Our scheduler also had to keep track of three pieces of state information: the queue of numbers to process, the results generated so far, and the list of active PIDs.

Most servers have a similar set of needs, so OTP provides libraries that do all the low-level work for us.

When we write an OTP server, we write a module containing one or more

callback functions with standard names. OTP will invoke the appropriate callback to handle a particular situation. For example, when someone sends a request to our server, OTP will call our handle_call function, passing in the request, the caller, and the current server state. Our function responds by returning a tuple containing an action to take, the return value for the request, and an updated state.

State and the Single Server

Think back to our recursive Fibonacci code. Where did it keep all the intermediate results as it worked? It passed them to itself, recursively, as parameters. In fact, all three of its parameters were used for state

information.

Now think about servers. They use recursion to loop, handling one request on each call. So they can also pass state to themselves as a parameter in this recursive call. And that's one of the things OTP manages for us. Our handler functions get passed the current state (as their last parameter), and they return (among other things) a potentially updated state. Whatever state a function returns is the state that will be passed to the next request handler.

Our First OTP Server

Let's write what is possibly the simplest OTP server. You pass it a number when you start it up, and that becomes the current state of the server. When you

call it with a :next_number request, it returns that current state to the caller, and at the same time increments the state, ready for the next call. Basically, each time you call it you get an updated sequence number.

Create a New Project Using Mix

Start by creating a new mix project in your work directory. We'll call it sequence.

Create the Basic Sequence Server

Now we'll create Sequence.Server, our server module. Move into the sequence directory, and create a subdirectory under lib/ also called sequence.

Add the file server.ex to lib/sequence/:

otp-server/1/sequence/lib/sequence/server.ex

The first thing to note is line 2. The use line effectively adds the OTP *GenServer* behavior to our module. This is what lets it handle all the callbacks. It also means we don't have to define every callback in our module—the behavior defines defaults for all but one of them.

The exception is the init/1 function, defined on line 4. You can think of init as being like the constructor in an object-oriented language: A constructor takes values and creates the object's initial state, and init takes some initial value and uses it to construct the state of

the server. This state is returned as the second element of the {:ok, state} tuple. In our case, we use the init function to set the initial value of our counter.

When a client calls our server, GenServer invokes its handle_call function. This function receives three parameters:

1. The information the client passed to the call
2. The PID of the client
3. The server state

Your implementation of the function should perform the actions associated with the first parameter, and may update the state (the third parameter). When the handle_call function exits, it must return the state (updated or not).

The initial state of a GenServer is set by the return value of the init function.

Our implementation is simple: we return a tuple to OTP.

The reply element tells OTP to reply to the client, passing back the value that is the second element. Finally, the tuple's third element defines the new state. This will be passed as the last parameter to handle_call the next time it is invoked.

Fire Up Our Server Manually

We can play with our server in IEx. Open it in the project's main directory, remembering the -S mix option.

We're using two functions from the Elixir GenServer module. The start_link function behaves like the spawn_link function we used in the previous chapter. It asks GenServer to start a new process and link to us (so we'll get notifications if it fails). We pass in the module to run as a server: the initial state (100 in this case). We could also pass GenServer options as a third parameter, but the defaults work fine here.

We get back a status (:ok) and the server's PID. The call function takes this PID and calls the handle_call function in the server. The call's second parameter is passed as the first argument to handle_call.

In our case, the only value we need to pass is the identity of the action we want to perform, :next_number. If you look at the definition of handle_call in the server, you'll see that its first parameter is :next_number. When Elixir invokes the function, it pattern-matches the argument in the call with this first parameter in the function. A server can support multiple actions by implementing multiple handle_call functions with different first parameters.

If you want to pass more than one thing in the call to a server, pass a tuple. For example, our server might need a function to reset the count to a given value. We could define the handler as

and call it with

Similarly, a handler can return multiple values by packaging them into a tuple or list.

Your Turn

- Exercise: OTP-Servers-1

  You're going to start creating a server that implements a stack. The call that initializes your stack will pass in a list of the initial stack contents.

  For now, implement only the `pop` interface. It's acceptable for your server to crash if someone tries to pop from an empty stack.

  For example, if initialized with `[5,"cat",9]`, successive calls to `pop` will return `5`, `"cat"`, and `9`.

One-Way Calls

The `call` function calls a server and waits for a reply. But sometimes you won't want to wait because there is no reply coming back. In those circumstances, use the GenServer `cast` function. (Think of it as casting your

request into the sea of servers.)

Just like call is passed to handle_call in the server, cast is sent to handle_cast. Because there's no response possible, the handle_cast function takes only two parameters: the call argument and the current state. And because it doesn't want to send a reply, it will return the tuple {:noreply, new_state}.

Let's modify our sequence server to support an :increment_number function. We'll treat this as a cast, so it simply sets the new state and returns.

otp-server/1/sequence/lib/sequence/server.ex

Notice that the cast handler takes a tuple

as its first parameter. The first element is :increment_number, and is used by pattern matching to select the handlers to run. The second element of the tuple is the delta to add to our state. The function simply returns a tuple, where the state is the previous state plus this number.

To call this from our IEx session, we first have to recompile our source. The r command takes a module name and recompiles the file containing that module.

Even though we've recompiled the code, the old version is still running. The VM doesn't hot-swap code until

you explicitly access it by module name. So, to try our new functionality we'll create a new server. When it starts, it will pick up the latest version of the code.

Tracing a Server's Execution

The third parameter to start_link is a set of options. A useful one during development is the debug trace, which logs message activity to the console.

We enable tracing using the debug option:

See how it traces the incoming call and the response we send back. A nice touch

is that it also shows the next state.

We can also include :statistics in the debug list to ask a server to keep some basic statistics:

Most of the fields should be fairly obvious. Timestamps are given as {{y,m,d},{h,m,s}} tuples. The reductions value is a measure of the amount of work the server does. It is used in process scheduling as a way of making sure all processes get a fair share of the available CPU.

The Erlang sys module is your interface to the world of *system messages*. These are sent in the background between

processes—they're a bit like the backchatter in a multiplayer video game. While two players are engaged in an attack (their real work), they can also be sending each other background messages: "Where are you?," "Stop moving," and so on.

The list associated with the debug parameter you give to GenServer is simply the names of functions to call in the sys module. If you say [debug: [:trace, :statistics]], then those functions will be called in sys, passing in the server's PID. Look at the documentation for sys to see what's available.[34]

This also means you can turn things on and off *after* you have started a server. For example, you can enable tracing on

an existing server using the following:

get_status is another useful sys function:

This is the default formatting of the status message GenServer provides. You have the option to change the data: part to a more application-specific message by defining a format_status function. This receives an option describing why the function was called, as well as a list containing the server's process dictionary and the current state. (Note that in the code that follows, the string *State* in the response is in single quotes.)

otp-server/1/sequence/lib/sequence/server.ex

If we ask for the status in IEx, we get the new message (after restarting the server):

Your Turn

- Exercise: OTP-Servers-2

  Extend your stack server with a push interface that adds a single value to the top of the stack. This will be implemented as a cast.

  Experiment in IEx with pushing and popping values.

GenServer Callbacks

GenServer is an OTP protocol. OTP works by assuming that your module defines a number of callback functions (six, in the case of a GenServer). If you were writing a GenServer in Erlang, your code would have to contain implementations of all six.

When you add the line ‘use GenServer‘ to a module, Elixir creates default implementations of these six callback functions. All we have to do is override the ones where we add our own application-specific behavior. Our examples so far have used the three callbacks ‘init‘, ‘handle_call‘, and

‘handle_cast‘. Here’s a full list:

init(start_arguments)

Called by GenServer when starting a new server. The parameter is the second argument passed to start_link. It should return {:ok, state} on success, or {:stop, reason} if the server could not be started.

You can specify an optional timeout using {:ok, state, timeout}, in which case GenServer sends the process a :timeout message whenever no message is received in a span of *timeout* ms. (The message is passed to the handle_info function.)

The default GenServer implementation sets the server state to the argument you pass.

handle_call(request, from, state)

Invoked when a client uses GenServer.call(pid, request). The from parameter is a tuple containing the PID of the client and a unique tag. The state parameter is the server state.

On success it returns {:reply, result, new_state}. The list that follows this one,, shows other valid responses.

The default implementation stops the server with a :bad_call error, so you'll need to implement

handle_call for every call request type your server implements.

handle_cast(request, state)

Called in response to GenServer.cast(pid, request).

A successful response is {:noreply, new_state}. It can also return {:stop, reason, new_state}.

The default implementation stops the server with a :bad_cast error.

handle_info(info, state)

Called to handle incoming messages that are not call or cast requests. For example, timeout messages are handled here. So are termination messages from any

linked processes. In addition, messages sent to the PID using send (so they bypass GenServer) will be routed to this function.

terminate(reason, state)

Called when the server is about to be terminated. However, as we'll discuss in the next chapter, once we add supervision to our servers, we don't have to worry about this.

code_change(from_version, state, extra)

Updates a running server without stopping the system. However, the new version of the server may represent its state differently from the old version. The code_change callback is invoked to change

from the old state format to the new.

format_status(reason, [pdict, state])

Used to customize the state display of the server. The conventional response is [data: [{'State', state_info}]].

The call and cast handlers return standardized responses. Some of these responses can contain an optional :hibernate or timeout parameter. If hibernate is returned, the server state is removed from memory but is recovered on the next request. This saves memory at the expense of some CPU. The timeout option can be the atom :infinite (which is the default) or a number. If the latter, a :timeout message is sent if the

server is idle for the specified number of milliseconds.

The first two responses are common between call and cast.

{ :noreply, new_state [, :hibernate | timeout] }

{ :stop, reason, new_state }

Signal that the server is to terminate.

Only handle_call can use the last two.

{ :reply, response, new_state [, :hibernate | timeout] }

Send response to the client.

{ :stop, reason, reply, new_state }

Send the response and signal that

the server is to terminate.

[illegible]Sequence.Server,

Naming a Process

The idea of referencing processes by their PIDs gets old quickly. Fortunately, there are a number of alternatives.

The simplest is local naming. We assign a name that is unique for all OTP processes on our node, and then we use that name instead of the PID whenever we reference it. To create a locally named process, we use the name: option when we start the server:

Tidying Up the Interface

As we left it, our server works but is ugly to use. Our callers have to make explicit GenServer calls, and they have to know the registered name for our server process. We can do better. Let's wrap this interface in a set of three functions in our server module: start_link, next_number, and increment_number. The first of these calls the GenServer start_link method. As we'll see in the next chapter, the name start_link is a convention. start_link must return the correct status values to OTP; as our code simply delegates to the GenServer module, this is taken care of.

Following the definition of start_link, the next two functions are the external API to issue call and cast requests to the running server process.

We'll also use the name of the module as our server's registered local name (hence the name: _MODULE_ when we start it, and the _MODULE_ parameter when we use call or cast).

otp-server/2/sequence/lib/sequence/server.ex

When we run this code in IEx, it's a lot cleaner:

This is the pattern that just about everyone uses in the Elixir world.

I have a different view. However, it certainly isn't mainstream, so feel free to skip the next section if you want to avoid becoming tainted by my heresies.

Making Our Server into a Component

Earlier I said that what Elixir calls an application, most people would call a component or a service. That's certainly what our sequence server is: a freestanding chunk of code that enjoyed generating successive numbers.

Despite being the canonical way of writing this, I don't like my implementation. It puts three things into a single source file:

- The API
- The logic of our service (adding one)
- The implementation of that logic in a server

Have another look at the code here. If

you didn't know what it did, how would you find out? Where's the code that does the component's logic? (The image gives you a hint.) It isn't obvious, and this is just a trivial service. Imagine working with a really complex one, with lots of logic.



```
defmodule Sequence.Server do
  use GenServer

  #####
  # External API

  def start_link(current_number) do
    GenServer.start_link(__MODULE__, current_number, name: __MODULE__)
  end

  def next_number do
    GenServer.call __MODULE__, :next_number
  end

  def increment_number(delta) do
    GenServer.cast __MODULE__, {:increment_number, delta}
  end

  #####
  # GenServer implementation

  def init(initial_number) do
    { :ok, initial_number }
  end

  def handle_call(:next_number, _from, current_number) do
    { :reply, current_number, current_number+1 }
  end

  def handle_cast({:increment_number, delta}, current_number) do
    { :noreply, current_number + delta }
  end

  def format_status(_reason, [ _pdict, state ]) do
    [data: [{'State', 'My current state is '#{inspect state}', and I'm happy'}]]
  end
end
```

That's why I'm experimenting with splitting the API, implementation, and server into three separate files.

We'll start afresh:

We'll put the API in the top-level lib/sequence.ex module, and the implementation and server in the two lower-level modules.

The API is the public face of our component. It is simply the top half of the previous server module:

otp-server/3/sequence/lib/sequence.ex

This forwards calls on to the server implementation:

otp-server/3/sequence/lib/sequence/server.ex

Unlike the previous server, this code contains no "business logic" (which in

our case is adding either 1 or some delta to our state). Instead, it uses the implementation module to do this:

otp-server/3/sequence/lib/sequence/impl.ex

Now, you're probably looking at this and thinking, "all that work just to implement a counter?" And you'd be right. But this chapter isn't about implementing a counter. It's all about implementing real-world servers. Because Elixir makes it easy to bundle all the code for a server into one module, most people do, and then they end up with some fairly highly coupled (and hard-to-test) code.

So think of this example, but with some

real, complex business logic. Imagine how you might write it.

You'd probably start with an API, just to see what it would look like. Then you might want to write and test some of the business logic. So go into the implementation module and do just that. What's more, test that code directly as you write it: no need to run it inside a server for that.

As you progress with the implementation, you may learn things that require changes to the overall API. Feel free.

Then, when you're feeling good about the code, add a server module and have the API use it.

The thing I like about this approach is that it leaves me a pure implementation of the actual logic on my component, independent of whether I choose to deploy it as a server. I can use (and test) the logic either as direct function calls or indirectly via a server.

Your Turn

- Exercise: OTP-Servers-3

  Give your stack server process a name, and make sure it is accessible by that name in IEx.

- Exercise: OTP-Servers-4

  Add the API to your stack module (the functions that wrap the GenServer calls).

- Exercise: OTP-Servers-5

  Implement the terminate callback in your stack handler. Use IO.puts to report the arguments it receives.

  Try various ways of terminating your server. For example, popping an empty stack will raise an exception. You might add code that calls System.halt(n) if the push handler receives a number less than 10. (This will let you generate different return codes.) Use your imagination to try different scenarios.

An OTP GenServer is just a regular

Elixir process in which the message handling has been abstracted out. The GenServer behavior defines a message loop internally and maintains a state variable. That message loop then calls out to various functions that we define in our server module: handle_call, handle_cast, and so on.

We also saw that GenServer provides fairly detailed tracing of the messages received and responses sent by our server modules.

Finally, we wrapped our message-based API in module functions, which gives our users a cleaner interface and decouples them from our implementation.

But we still have an issue if our server crashes. We'll deal with this in the next chapter, when we look at supervisors.

Footnotes

[34] http://www.erlang.org/documentation/doc-5.8.3/lib/stdlib-1.17.3/doc/html/sys.html

Copyright © 2018, The Pragmatic Bookshelf.

Chapter 18

OTP: Supervisors

I've said it a few times now: the *Elixir way* says not to worry much about code that crashes; instead, make sure the overall application keeps running.

This might sound contradictory, but really it is not.

Think of a typical application. If an unhandled error causes an exception to be raised, the application stops. Nothing else gets done until it is restarted. If it's a server handling multiple requests, they all might be lost.

The issue here is that one error takes the whole application down.

But imagine that instead your application consists of hundreds or thousands of processes, each handling just a small part of a request. If one of those crashes, everything else carries on. You might lose the work it's doing, but you can design your applications to minimize even that risk. And when that process gets restarted, you're back running at 100%.

In the Elixir and OTP worlds, *supervisors* perform all of this process monitoring and restarting.

Supervisors and Workers

An Elixir supervisor has just one purpose—it manages one or more processes. (As we'll discuss later, these processes can be workers or other supervisors.)

At its simplest, a supervisor is a process that uses the OTP supervisor behavior. It is given a list of processes to monitor and is told what to do if a process dies, and how to prevent restart loops (when a process is restarted, dies, gets restarted, dies, and so on).

To do this, the supervisor uses the Erlang VM's process-linking and -

monitoring facilities. We talked about these when we covered spawn.

You can write supervisors as separate modules, but the Elixir style is to include them inline. The easiest way to get started is to create your project with the --sup flag. Let's do this for our sequence server.

The only apparent difference is the appearance of the file lib/sequence/application. Let's have a look inside (I stripped out some comments…):

Our start function now creates a supervisor for our application. All we

need to do is tell it what we want supervised. Copy the second version of the Sequence.Server module[35] from the last chapter into the lib/sequence folder. Then uncomment and change the line in the child_list to reference this module:

otp-supervisor/1/sequence/lib/sequence/application.ex

Let's look at what's going to happen:

- When our application starts, the start function is called.
- It creates a list of child server modules. In our case, there's just one, the Sequence.Server. Along with the module name, we specify an argument to be passed to the server when we start it.
- We call Supervisor.start_link, passing it the list of child specifications and a set of options. This creates a supervisor process.
- Now our supervisor process calls the start_link

function for each of its managed children. In our case, this is the function in Sequence.Server. This code is unchanged—it calls GenServer.start_link to create a GenServer process.

Now we're up and running. Let's try it:

So far, so good. But the key thing with a supervisor is that it is supposed to manage our worker process. If it dies, for example, we want it to be restarted. Let's try that. If we pass something that isn't a number to increment_number, the process should die trying to add it to the current number.

We get a wonderful error report that shows us the exception, along with a stack trace from the process. We can

also see the message we sent that triggered the problem.

But when we then ask our server for a number, it responds as if nothing had happened. The supervisor restarted our process for us.

This is excellent, but there's a problem. The supervisor restarted our sequence process with the initial parameters we passed in, and the numbers started again from 123. A reincarnated process has no memory of its past lives, and no state is retained across a crash.

Your Turn

- Exercise: OTP-Supervisors-1

  Add a supervisor to your stack application. Use IEx to make sure it starts the server correctly. Use the server normally, and then crash it (try popping from an empty stack). Did it restart? What were the stack contents after the restart?

Managing Process State Across Restarts

Some servers are effectively stateless. If we had a server that calculated the factors of numbers or responded to network requests with the current time, we could simply restart it and let it run.

But our server is not stateless—it needs to remember the current number so it can generate an increasing sequence.

All of the approaches to this involve storing the state outside of the process. Let's choose a simple option—we'll write a separate process that can store and retrieve a value. We'll call it our *stash*. The sequence server can store its current number in the stash whenever it terminates, and then we can recover the number when we restart.

Now, we have to think about lifetimes. Our sequence server should be fairly robust, but we've already found one thing that crashes it. In actuarial terms, it isn't the fittest process in the scheduler queue. But our stash process must be more robust—it must outlive the sequence server, at the very least.

We have to do two things to make this

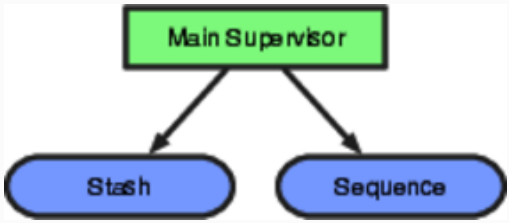
happen. First, we make it as simple as possible. The fewer moving parts in a chunk of code, the less likely it is to go wrong.

Second, we have to arrange things so that the supervisor isolates it from failures in the sequence server.

Let's do the first of these things now. We'll create a trivial server whose entire purpose is to store a single value.

otp-supervisor/2/sequence/lib/sequence/stash.ex

Now we want to supervise it. It'll be running alongside the sequence server:



This is the first time we've had two servers supervised together. So now we have to face a question: what happens if just one of them crashes? The answer depends on the *supervision strategy* we have chosen.

:one_for_one

if a server dies, the supervisor will restart it. This is the default strategy.

:one_for_all

if a server dies, all the remaining servers are first terminated, and then the servers are all restarted.

:rest_for_one

> if a server dies, the servers that follow it in the list of children are terminated, and then the dying server and those that were terminated are restarted.

Right now we only have two servers. The stash server is supposed to be eternal, while the sequence server might crash and need restarting. Because of this, we know that we can't use :one_for_all.

This leaves two choices, and both would work. If we use :one_for_one, a failing sequence server will restart, and the stash will not be affected. If we use :rest_for_one, the same thing will happen, but only if the sequence server

follows the stash in the list of children.

Which to choose? I vote for :rest_for_one, not because it has any different behavior to :one_for_one, but because I feel it expresses my intent better. A :rest_for_one supervision strategy explicitly says, “this server depends on the health of previous servers in the list.”

Let’s add the stash and update the supervision strategy in our supervisor startup code:

otp-supervisor/2/sequence/lib/sequence/application.ex

Finally, we need to change our sequence server to use this stash. For now, we’ll

have it set its state to the current value in the stash when it starts, and have it store the value back into the stash if it crashes.

Setting the initial state simply means fetching the current value from the stash in our sequence server's init function. Handling the sequence server exiting involves writing another callback, terminate. Here's the full code:

otp-supervisor/2/sequence/lib/sequence/server.ex

Let's try this change in IEx:

How cool is that? The server code

crashed, but was then restarted automatically. And, in the process, the state was stored away in the stash and then recovered—the sequence continued uninterrupted.

Your Turn

- Exercise: OTP-Supervisors-2

  Rework your stack server to use a supervision tree with a separate stash process to hold the state. Verify that it works and that when you crash the server state is retained across a restart.

Simplifying the Stash

The sole job of our stash module is to store a value. When we look at agents, we'll see that they're a perfect fit for this, and we'll be able to simplify our code.

Worker Restart Options

So far we've looked at supervision from the point of view of the supervisor. In particular, we've seen how the supervision strategy tells the supervisor how to deal with the death of a child process.

There's a second level of configuration that applies to individual workers. The most commonly used of these is the :restart option.

Previously we said that a supervisor strategy (such as :one_for_all) is invoked when a worker dies. That's not strictly true. Instead, the strategy is invoked

when a worker needs restarting. And the conditions when a worker should be restarted are dictated by its `restart:` option:

`:permanent`

This worker should always be running—it is permanent. This means that the supervision strategy will be applied whenever this worker terminates, for whatever reason.

`:temporary`

This worker should never be restarted, so the supervision strategy is never applied if this worker dies.

`:transient`

> It is expected that this worker will at some point terminate normally, and this termination should not result in a restart. However, should this worker die abnormally, then it should be restarted by running the supervision strategy.

The simplest way to specify the restart option for a worker is in the worker module. You add it to the use GenServer (or use Supervisor) line:

A Little More Detail

You don't have to know the information in this section first time around, but as someone always asks….

We've seen that you start a supervisor by passing it a list of children. Just what is that list?

At the very lowest level, it is a list of *child specifications*. A child spec is an Elixir map. It describes which function to call to start the worker, how to shut the worker down, the restart strategy, the worker type, and any modules apart from the main module that form part of the worker.

You can create a child spec map using the Supervisor.child_spec/2 function.

At the next level up, you can specify a worker by giving its module name (or a tuple containing the module and the initial arguments). In this case, the

supervisor assumes you've implemented a child_spec function in that module, and calls that function to get the specification.

Going up one more level, when you add the line use GenServer to a server module, Elixir will define a default child_spec function in that module. This function by default returns a map that tells the supervisor that the start function will be start_link and that the restart strategy will be :permanent. You can override these defaults with the options you give use GenServer.

In practice, the option you'll change the most will be :restart. Although :permanent is a good default for long-running servers, it won't work for

servers that do a job and then exit. These types of servers should have a restart value of :transient.

Supervisors Are the Heart of Reliability

Think about our previous example; it was both trivial and profound. It was trivial because there are many ways of achieving some kind of fault tolerance with a library that returns successive numbers. But it was profound because it is a concrete representation of the idea of building rings of confidence in our code. The outer ring, where our code interacts with the world, should be as reliable as we can make it. But within that ring there are other, nested rings. And in those rings, things can be less than perfect. The trick is to ensure that the code in each ring knows how to deal with failures of the code in the next ring

down.

And that's where supervisors come into play. In this chapter we've seen only a small fraction of supervisors' capabilities. They have different strategies for dealing with the termination of a child, different ways of terminating children, and different ways of restarting them. There's plenty of information online about using OTP supervisors.

But the real power of supervisors is that they exist. The fact that you use them to manage your workers means you are forced to think about reliability and state as you design your application. And that discipline leads to applications with very high availability—in *Programming*

Erlang (2nd edition) [Arm13], Joe Armstrong says OTP has been used to build systems with 99.9999999% reliability. That's nine nines. And that ain't bad.

(In case you were wondering, that equates to a complete application outage of roughly 1 second every 30 years. I don't know how you'd even measure that, which makes me a little suspicious….)

There's one more level in our lightning tour of OTP—the application. But before we look at that, let's use what we learned so far and build some real-world code.

Footnotes

[35] http://media.pragprog.com/titles/elixir16/code/otp-server/2/sequence/lib/sequence/server.ex

Copyright © 2018, The Pragmatic Bookshelf.

Chapter 19

A More Complex Example

When I first used Elixir, I struggled with how to organize my applications. When should I use servers? How do supervisors fit in? Even basic questions such as “how many applications should I write?” made me nervous.

Frankly, I’m still thinking about these questions, three years later, in the same way I’m still thinking about object-oriented system design after 30 years of doing it. But, on the Elixir front, I have come up with an approach that helps me

think through these issues.

It isn't rocket science. I just ask myself these five questions:

- What is the environment and what are its constraints?
- What are the obvious focal points?
- What are the runtime characteristics?
- What do I protect from errors?
- How do I get this thing running?

What I'm going to show in this chapter is just my ad hoc approach. Please don't take it as any kind of "methodology." But if, like me, you sometimes feel overwhelmed when designing a new Elixir system, these steps might help.

Let's write a simple application to show you what I mean.

Introduction to Duper

I have loads of duplicate files littering my computers. In an effort to tame this, let's write a duplicate-file finder. Well call it *Duper* (so we can later create a paid version called *SuperDuper*). It'll work by scanning all the files in a directory tree, calculating a hash for each. If two files have the same hash, we'll report them as duplicates.

Let's start asking the questions.

Q1: What is the environment and what are its constraints?

We're going to run this on a typical computer. It'll have roughly two orders of magnitude more file storage than

main memory. Files will range in size from 10^0 to 10^{10} bytes, and there will be roughly 10^7 of them.

What this means:

We need to allow for the fact that although we have to load files into memory to determine their hashes, it is possible we won't have enough memory to load the largest files in whole. We definitely will not be able to process all the files at once.

It also means that our design will need to cater to both big and small files. Big files will take more time to read into memory than small files, and they will also take longer to hash.

Q2: What are the focal points?

A focal point represents a responsibility of the application. By considering the focal points now, we can start to reduce coupling in the application as a whole: each focal point can be tightly coupled internally but loosely coupled to the others. This coupling can be both structural (for example, the representation of data) and temporal (for example, the sequence in which things will happen).

In Duper, we can fairly easily identify some key focal points:

- We need to have a place where we collect results. We are calculating a hash value for each file, so this results store will need to hold all of these. As we are looking for duplicate hashes, it would make sense for this to be some kind of key/value store internally, where the key is the hash and the value is a list of all files with that hash. However, this is

an implementation detail, and the implementation shouldn't leak through our API.

- We need to have something that can traverse the filesystem, returning each path just once.
- We need to have something that can take a path and calculate the hash of the corresponding file. Because an individual file may be too big to fit in memory, we'll have to read it in chunks, calculating the hash incrementally.
- Because we know we will need to be processing multiple files concurrently in order to maximize our use of the CPU and IO bandwidth, we'll need something that orchestrates the overall process.

This list may well change as we start to write code, but it's good enough to get us to the next step.

What this means:

At the very least, each focus we identify is an Elixir module. My experience is that it's wise to assume that most if not

all are going to end up being servers. I'm also coming to believe that many should even be separate Elixir applications, but that's not something I'm going to dig into here.

Our code will be structured into four servers. Although we could do it with fewer, using four means we can have specific characteristics for each. The four are as follows:

- The *Results*. This is the most important server, as it holds the results of the scanning in memory. We need it to be reliable, so we won't put much code in it.

- The *PathFinder*. This is responsible for returning the paths to each file in the directory tree, one at a time.

- The *Worker*. This asks the PathFinder for a path, calculates the hash of the resulting file's contents, and passes the result to the gatherer.

- The *Gatherer*. This is the server that both starts the ball rolling and determines when things have completed. When they do, it fetches the results and reports on them.

Q3: What are the runtime characteristics?

Our application is going to spend the vast majority of its time in the workers, as this is where we read the files and calculate the hash values. Our goal is to keep both the processors and the IO bus as busy as possible in order to maximize performance.

If we have just one worker, then it would read a file, hash it, read the next, hash it, and so on. We'd alternate between being IO bound and CPU bound. This doesn't come close to maximizing our performance.

On the other hand, if we had one worker for each file, then they could be reading and hashing at the same time. However, we'd run out of memory on our machine, as we'd effectively be trying to load our filesystem into memory.

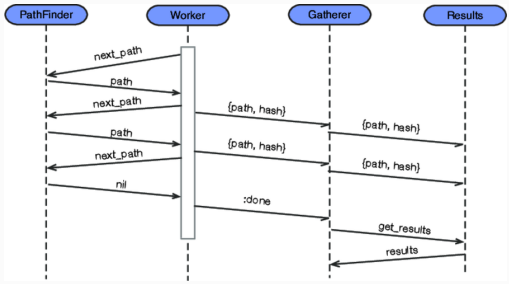
The sweet spot lies in between.

One approach is to create *n* workers, and then divide the work equally between them. This is the typical push model: plan the work up front and let it execute. The problem with this approach is that it assumes that each file is about the same size. If that's not the case (and on my machine it certainly isn't), then it would be possible to give one worker mostly small files and another mostly large files. The first would finish early, and

would then sit idle while the second chewed through its workload.

The approach I prefer in this scenario is what I call *hungry consumer*. It's a pull model, where each worker asks for the next thing to do, processes it, and then asks for more work. In this scheme a worker that has a small file to process will get it done quickly, then ask for more work. One with a bigger file will take more time. There'll never be an idle worker until we get to the very last files.

The following sequence diagram shows how messages flow in this system. Notice that we have a mixture of synchronous messaging (the pairs of arrows going in opposite directions) and asynchronous messaging.



Q4: What do I protect from errors?

This is where we get to be pragmatic!

In an ideal world, nothing will fail, and everything should be protected from errors.

The real world is different—we can only do so much protecting. How much do we need? It depends. If we're writing software for a pacemaker, then I'd suggest the vast majority of the implementation effort should go into error protection. If we're writing a duplicate-file finder, then not so much.

We can assume that running our finder across 500 GB of files will take minutes, not seconds. That means that if we have a failure reading one file, we don't want to stop the whole application and lose the work done so far—it is good enough to continue ignoring that file.

What this means:

We want to protect the accumulating results, but we're not as worried about the individual workers—workers can simply restart on failure and process the next file.

Q5. How do I get this thing running?

Sequential programs are easy to start: you just run them. Applications that have many moving parts are more complex: you have to make sure that the

various servers are started so that if server A needs to call server B, then B is running before A makes that call.

How should our four servers be started?

The sequence diagram tells us most of the answer.

- Worker depends on PathFinder and Gatherer.
- Gatherer depends on Results.
- Pathfinder and Results depend on nothing.

What this means:

Pathfinder and Results should be started first, followed by Gatherer, and then by the workers.

In terms of implementation, this is fairly straightforward. We simply list the servers in this order as children of a

supervisor, and the supervisor will make sure each child is running before starting the next.

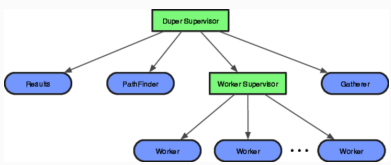
But we also know that we will have multiple workers. Rather than try to find a way of creating them up at the top level, we'll add something new: a subsupervisor. This subsupervisor is responsible for just the pool of workers.

Adding this supervisor opens up an interesting possibility. If all the children of a supervisor are the same, then that supervisor can be used to create them dynamically. Our gatherer server could create a pool of workers when it kicks off the application. This would let us experiment with the effect the number of workers has on elapsed time.

Our new dependency structure looks like this:

- Worker depends on PathFinder and Gatherer.
- Gatherer depends on Results and the worker supervisor.
- Pathfinder and Results depend on nothing.

This gives us a supervision structure that looks like this:



Of course, all this is just theory. Let's start getting some feedback by writing code.

The Duper Application

We'll start by creating a supervised application:

Time to start writing servers.

The Results Server

The results server wraps an Elixir map. When it starts, it sets its state to an empty map. The keys of this map are hash values, and the values are the list of one of more paths whose files have that hash.

The server provides two API calls: one to add a hash/path pair to the map, the

second to retrieve entries that have more than one path in the value (as these are two duplicate files).

This is similar to the code we wrote for the sequence stash:

duper/1/duper/lib/duper/results.ex

The only mild magic in this code is the use of Map.update/4. This wonderful function takes a map, a key, an initial value, and a function. If the key is not present in the map, then a new map is returned with that key and initial value added. If the key *is* present, then the corresponding value is passed to the function, and whatever the function returns becomes the updated value in the

returned map. In our case, we're using it to create a single-element path list the first time a hash is encountered, and then to add paths to that list on duplicates.

We'll add this server to the list of top-level children in application.ex.

This code is easy to test:

duper/1/duper/test/duper/results_test.exs

The PathFinder Server

Our next server is responsible for returning all the file paths in a filesystem tree, one at a time.

Elixir doesn't have a filesystem-traversal API built in, so we look on 'hex.pm' and find dir_walker,[36] which we just need to wrap in a trivial GenServer whose state is the directory walker's PID. So we add the dependency to our mix.exs file:

duper/1/duper/mix.exs

and code the server in lib/duper/path_finder.ex:

duper/1/duper/lib/duper/path_finder.ex

Finally we add the pathfinder server to our application's list of children:

Notice we used a tuple to specify the PathFinder server. That's because it requires us to pass in the root of the tree to be searched as a parameter. Here, I'm using the current working directory, ".", which will work well for playing with the code.

The Worker Supervisor

When we investigated how we'd get things started,, we realized that we'd need a different supervisor for our workers. This supervisor will only manage the worker servers, and it will let us add servers dynamically, after the application has started.

The simplest way to do this is to use a *DynamicSupervisor*. This type of supervisor allows you to create an

arbitrary number of workers at runtime. (A DynamicSupervisor encapsulates what used to be the :simple_one_for_one strategy in regular supervisors. You can still do it the old way, but DynamicSupervisors let you express your intent better.)

Let's create the supervisor (in lib/duper/worker_supervisor.ex) and then see how it works.

duper/1/duper/lib/duper/worker_supervisor.ex

The supervisor is just a regular Elixir module. It starts with use DynamicSupervisor, which gives it its super(visor) powers.

The start_link function works the same in a supervisor as it does in a GenServer: it is called to start the server containing the supervisor code. Inside this server, Elixir automatically calls the init callback, which in turn initializes the supervisor code itself. This initialization receives the supervisor options. In the case of a dynamic supervisor, this can only be strategy: one_for_one.

Later, we can call the add_worker function. This calls the supervisor, telling it to add another child based on the child specification we pass. In this case, we tell it to start Duper.Worker. A new server is created for each call, and these servers run in parallel. As a result,

each time add_worker is called, a new Duper.Worker instance is spawned.

Note: One side effect of the fact that the same module is run in multiple child servers is we can't give the children a name in their start_link function. If we did, then there'd be multiple servers with the same name, which Elixir doesn't allow.

Let's remember to add the supervisor to the list of top-level children.

Thinking About Supervision Strategies

Whenever I add a child to a supervisor's list, I stop and think about the supervision strategy: how do I want the

failure of a child managed by this supervisor to affect the other children?

If the results server fails, then all is lost, and we have to restart everything. The same applies to the pathfinder: although we could in theory work out how far into the folder structure we were if it crashed, and restart from there, in practice that would be difficult, so for now we treat a failure of the pathfinder as a failure of the application.

What about the worker supervisor? Here we have to be careful. The worker supervisor handles the actual worker processes. If one of these fails, the worker supervisor simply restarts it and the application continues. But the failure of a worker does not mean that the

worker supervisor itself has failed. In fact, in the very unlikely event that the worker supervisor fails, it's probably best to assume we can't continue and stop the application.

So, for all three children we have on this top-level supervisor, a failure means the application should stop. The strategy that enforces this is :one_for_all, so we change our code accordingly.

That's all we're going to do on the worker side of things for now. Let's write the gatherer, and then circle back and implement the actual worker.

The Gatherer Server

Looking at the sequence diagram, we can see that the gatherer is invoked by the workers. Each worker can tell the gatherer that it has run out of work (by calling done()) or it can give it the results of hashing a file.

The gatherer has one more function, not shown in the diagram. It is responsible for starting the workers, and it is responsible for determining when the application has finished processing the files.

To do this, it maintains a simple state: the number of worker servers that are currently running.

Knowing that, we can write most of the gatherer server:

duper/1/duper/lib/duper/gatherer.ex

See how the implementation of :done keeps track of the number of running workers? As each signals it is done the count is decremented, until the last :done is received, where we report the results and exit.

We *could* try something like this:

However, this won't work, and it won't work in a fairly ugly way.

Remember when we described how supervisors started children in order? They wait for each child to initialize itself before starting the next.

In the preceding code, we're still initializing the gatherer when we start adding workers. The workers may start running before the initialization of the gatherer finishes. In this case, messages they send to it may well get lost. If you're only traversing a small filesystem tree, it is even possible that a worker might signal :done before the gatherer is ready, in which case the system will hang, because it will never know that it has finished.

The answer to this mess is to follow a simple rule: when you're initializing a server, don't interact with anything that uses that server.

So how do we get the workers running? The answer will be familiar to anyone

who has written JavaScript—we arrange for a callback into our gatherer server after initialization is complete:

Here the init function uses send_after to tell the runtime to queue a message to this server immediately (that is, after waiting 0 ms). When the init function exits, the server is then free to pick up this message, which triggers the handle_info callback, and the workers get started.

So, now that the gatherer code is ready, we just have to remember to start it:

duper/1/duper/lib/duper/application.ex

What About the Workers?

Referring back one last time to the sequence diagram, we can see that the workers are a little strange: they have no incoming API. All they do is ask for a path, compute the hash of the corresponding file, and send the hash to the gatherer. At some point, there are no paths left, so they then send a :done notification to the gatherer instead.

Here's the code:

duper/1/duper/lib/duper/worker.ex

Notice we use the same trick in the init() function to call back into ourselves, invoking handle_info(:do_one_file,_). This function asks the pathfinder for the

next file, and then passes the returned value to add_result().

If the pathfinder returns nil, it has run out of files, so we tell the gatherer that we're done. Otherwise, we call a private function to calculate the hash of the file contents, pass the path and the hash to the gatherer, and then send ourselves another :do_one_file message, causing the whole process to repeat.

One other thing to note—we flagged this server as being transient:

This means that the supervisor will not restart it if it terminates normally, but

will restart it if it fails.

But Does It Work?

Let's see where we are. We're implemented four GenServers and two supervisors. When the application starts, it will start the top-level supervisor, which in turn starts Results, PathFinder, WorkerSupervisor, and Gatherer.

When Gatherer starts (and it will start last), it tells the worker supervisor to start a number of workers. When each worker starts, it gets a path to process from PathFinder, hashes the corresponding file, and passes the result to Gatherer, which stores the path and the hash in the Results server. When there are no more files to process, each

worker sends a :done message to the gatherer. When the last worker is done, the gatherer reports the results.

Everything seems to be wired up. Let's try it:

Hmm…that's strange. No output.

The first time this happened to me, I wasted most of a day working it out. And the problem is obvious once you know what's happening.

The mix run command runs your application. Once it has it running, mix exits: mission accomplished.

But your application never finished; it

just got started and mix went away. We have to tell mix not to exit.

Much better. Even inside our Elixir project we have duplicated files, mostly between the test and dev environments.

Let's Play with Timing

Our lib/duper/application.ex file contains parameters that tell the app where to search and how many workers to use when searching. (We'll see in the next chapter how to move those values out of code and onto the command line.)

Let's change these parameters. My ~/Pictures folder used 30 GB to store about 6,000 old pictures from when I

used iPhoto. Let's look for duplicates in that folder with one worker, two workers, and so on, recording elapsed time.

Here are the parameters for using a single worker:

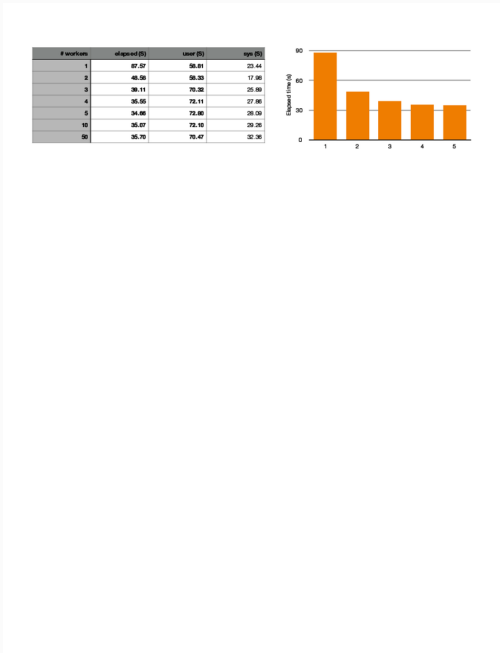
Run it:

We found 1,900-odd duplicated photos in about 88 seconds. The Elixir runtime used about 98% of one of my cores during this process.

Let's try with two workers. Alter application.ex, and run this:

Nice! It ran almost twice as fast. It means that I'm successfully overlapping the IO and the hashing.

To cut a long story short, here are the results for 1..5, 10, and 50 workers.



# workers	elapsed (S)	user (S)	sys (S)
1	87.57	55.81	23.44
2	48.58	58.33	17.96
3	39.11	70.32	25.89
4	35.55	72.11	27.98
5	34.66	72.66	28.09
10	35.07	72.10	29.26
50	35.70	70.47	32.36

As my machine has only two processors (four cores, but two are just

hyperthreading), that's about as good as I could expect.

Planning Your Elixir Application

This book is about thinking differently. We started by thinking about the code we write, and how a function style with immutable data forces us to think in terms of transformations.

In the last few chapters we've come across another dimension of this: thinking about how we structure our application. Our code is no longer monolithic. Instead we think about independent, interacting servers. (You might even call them services.)

This shift in thinking is a difficult one, because it involves both conceptual

adjustments and practical deployment issues. It is something you'll become more comfortable with over time. But remember to ask yourself the five questions, and your path should be easier:

- What is the environment and what are its constraints?
- What are the obvious focal points?
- What are the runtime characteristics?
- What do I protect from errors?
- How do I get this thing running?

Next Up

We have an application, but it doesn't really work well for us: things are hardcoded. Let's investigate a little more about what it means to be an application in the next chapter.

Footnotes

[36] https://hex.pm/packages/dir_walker

Copyright © 2018, The Pragmatic Bookshelf.

Chapter 20

OTP: Applications

So far in our quick tour of Elixir and OTP we've looked at server processes and the supervisors that monitor them. There's one more stage in our journey—the application.

This Is Not Your Father's Application

Because OTP comes from the Erlang world, it uses Erlang names for things. And unfortunately, some of these names are not terribly descriptive. The name *application* is one of these. When most of us talk about applications, we think of a program we run to do something—maybe on our computer or phone, or via a web browser. An application is a self-contained whole.

But in the OTP world, that's not the case. Instead, an application is a bundle of code that comes with a descriptor. That descriptor tells the runtime what dependencies the code has, what global

names it registers, and so on. In fact, an OTP application is more like a dynamic link library or a shared object than a conventional application.

It might help to see the word *application* in your head but pronounce it *component* or *service*.

For example, back when we were fetching GitHub issues using the HTTPoison library, what we actually installed was an independent application containing HTTPoison. Although it looked like we were just using a library, mix automatically loaded the HTTPoison application. When we then started it, HTTPoison in turn started a couple of other applications that it needed (SSL and Hackney), which in

turn kicked off their own supervisors and workers. And all of this was transparent to us.

I've said that applications are components, but some applications are at the top of the tree and are meant to be run directly.

In this chapter we'll look at both types of application component (see what I did there?). In reality they're virtually the same, so let's cover the common ground first.

The Application Specification File

You probably noticed that every now and then mix will talk about a file called *name*.app, where *name* is your application's name.

This file is called an *application specification* and is used to define your application to the runtime environment. Mix creates this file automatically from the information in mix.exs combined with information it gleans from compiling your application.

When you run your application this file is consulted to get things loaded.

Your application does not need to use

all the OTP functionality—this file will always be created and referred to. However, once you start using OTP supervision trees, stuff you add to mix.exs will get copied into the .app file.

application,

Turning Our Sequence Program into an OTP Application

So, here's the good news. The application here is already a full-blown OTP application. When mix created the initial project tree, it added a supervisor (which we then modified) and enough information to our mix.exs file to get the application started. In particular, it filled in the application function:

This says that the top-level module of our application is called Sequence. OTP assumes this module will implement a start function, and it will pass that function an empty list as a parameter.

In our previous version of the start function, we ignored the arguments and instead hard-wired the call to start_link to pass 123 to our application. Let's change that to take the value from mix.exs instead. First, change mix.exs to pass an initial value (we'll use 456):

Then change the application.ex code to use this passed-in value:

otp-app/sequence/lib/sequence/application.ex

We can check that this works:

Let's look at the application function

again.

The mod: option tells OTP the module that is the main entry point for our app. If our app is a conventional runnable application, then it will need to start somewhere, so we'd write our kickoff function here. But even pure library applications may need to be initialized. (For example, a logging library may start a background logger process or connect to a central logging server.)

For the sequence app, we tell OTP that the Sequence module is the main entry point. OTP will call this module's start function when it starts the application. The second element of the tuple is the parameter to pass to this function. In our case, it's the initial number for the

sequence.

There's a second option we'll want to add to this.

The registered: option lists the names that our application will register. We can use this to ensure each name is unique across all loaded applications in a node or cluster. In our case, the sequence server registers itself under the name Sequence.Server, so we'll update the configuration to read as follows:

otp-app/sequence/mix.exs

Now that we've done the configuring in mix, we run mix compile, which both compiles the app and updates the

sequence.app application specification file with information from mix.exs. (The same thing happens if we run mix using iex -S mix.)

Mix tells us it has created a sequence.app file, but where is it? You'll find it tucked away in _build/dev/lib/sequence/ebin. Although a little obscure, the directory structure under _build is compatible with Erlang's OTP way of doing things. This makes life easier when you release your code. You'll notice that the path has dev in it—this keeps things you're doing in development separate from other build products.

Let's look at the sequence.app that was generated.

otp-app/sequence/_build/dev/lib/sequence/ebin/sequence.app

This file contains an Erlang tuple that defines the app. Some of the information comes from the project and application section of mix.exs. Mix also automatically added a list of the names of all the compiled modules in our app (the .beam files) and a list of the apps our app depends on (kernel, stdlib, and elixir). That's pretty smart.

More on Application Parameters

In the previous example, we passed the integer 456 to the application as an initial parameter. Although that's

valid(ish), we really should have passed in a keyword list instead. That's because Elixir provides a function, Application.get_env, to retrieve these values from anywhere in our code. So we probably should have set up mix.exs with

and then accessed the value using get_env. We call this with the application name and the name of the environment parameter to fetch:

Your call.

Supervision Is the Basis of Reliability

Let's briefly recap. In that last example, we ran our OTP sequence application using mix. Looking at just our code, we see that a supervisor process and two worker processes got started. These were knitted together so our system continued to run with no loss of state even if the worker that we talked to crashed. And any other Erlang process on this node (including IEx itself) can talk to our sequence application and enjoy its stream of freshly minted integers.

You probably noticed that the start function takes two parameters. The

second corresponds to the value we specified in the mod: option in the mix.exs file (in our case, the counter's initial value). The first parameter specifies the status of the restart, which we're not going to get into, because…

Your Turn

- Exercise: OTP-Applications-1

  Turn your stack server into an OTP application.

- Exercise: OTP-Applications-2

  So far, we haven't written any tests for the application. Is there anything you can test? See what you can do.

Releasing Your Code

One way Erlang achieves nine-nines application availability is by having a rock-solid release-management system. Elixir makes this system easy to use.

Before we get too far, let's talk terminology.

A *release* is a bundle that contains a particular version of your application, its dependencies, its configuration, and any metadata it requires to get running and stay running. A *deployment* is a way of getting a release into an environment where it can be used.

A *hot upgrade* is a kind of deployment

that allows the release of a currently running application to be changed while that application continues to run—the upgrade happens in place with no user-detectable disruption.

In this section we'll talk about releases and hot upgrades. We won't dig too deeply into deployment.

Distillery—The Elixir Release Manager

Distillery is an Elixir package that makes most release tasks easy. In particular, it can take the complexity that is the source of your project, along with its dependencies, and reduce it down to a single deployable file.

Imagine you were managing the deployment of hundreds of thousands of lines of code into running telephone switches, while maintaining all the ongoing connections, providing a full audit trail, and maintaining contractual uptime guarantees. This is clearly complex. Very complex. And this is the task the Erlang folks faced, so they

created tools that help.

Distillery is a layer of abstraction on top of this complexity. Normally it manages to hide it, but sometimes the lower levels leak out and you get to see how the sausage is made.

This book isn't going to get that deep. Instead, I just want to give you a feel for the process.

Before We Start

In Elixir, we version both the application code and the data it operates on. The two are independent—we might go for a dozen code releases without changing any data structures.

The code version is stored in the project

dictionary in mix.exs. But how do we version the data? Come to think of it, where do we even define the data?

In an OTP application, all state is maintained by servers, and each server's state is independent. So it makes sense to version the app data within each server module. Perhaps a server initially holds its state in a two-element tuple. That could be version 0. Later, it is changed to hold state in a three-element tuple. That could be version 1.

We'll see the significance of this later. For now, let's just set the version of the state data in our server. We use the @vsn (version) directive:

otp-app/sequence_v0/lib/sequence/server.ex

Now let's generate a release.

Your First Release

First, we have to add distillery as a project dependency. Open the sequence project's mix.exs file and update the deps function.

otp-app/sequence_v0/mix.exs

(The runtime: false option tells mix that Distillery is not to be started with the running application.)

Remember to install the dependency:

Distillery makes sensible choices for the

various configuration options, so for a basic app like this we're now ready to create our first release. To start off we create the release configuration:

If you're following along at home, have a look at rel.config.exs (it's too large to show here). The defaults look good, so let's build the actual release. We'll tell it we want a production version of the release. If we don't, the release it generates will be for development, and won't be a self-contained package.

Distillery got the application name and version number from your mix.exs file, and packaged your app into the _build/

dev/rel/ directory:

The most important file is rel/sequence/releases/0.0.1/sequence.tar.gz. It contains everything needed to run this release. This is the file we deploy to our servers.

A Toy Deployment Environment

I don't want to slow things down by having you provision a server in the cloud, so I'm going to deploy to my local machine. However, to make it a little more realistic, I'll pretend this machine is remote, and use ssh to do all the deploying. I'll also be creating directories and copying files manually.

In practice, you'd want to automate all of this with something like Capistrano or Ansible.

We'll store the releases in a deploy directory. I'll put this inside my home directory—feel free to put it anywhere (writable) you want.

Deploy and Run the App

Now we need to set up the initial release and its directory structure. Copy the sequence.tar.gz file into the deploy directory, and then extract its contents.

The app is now ready to run. The scripts in deploy/bin control it. These, in turn,

delegate to scripts in the current release directory (all on the server).

Let's start an IEx console. (The ssh -t option lets us control the remote IEx with ^C and ^G.)

(Leave this session running—we'll use it to demonstrate hot code reloading.)

A Second Release

Our marketing team ran a focus group. It seems our customers want the next_number function to return a message like "the next number is 458."

First we'll change server.ex:

otp-app/sequence_v1/lib/sequence/server.ex

Then we'll bump the application's version number in mix.exs.

otp-app/sequence_v1/mix.exs

(We don't have to change the @vsn value—the representation of the server's state is not affected by this change.)

After exhaustive testing, we decide we're ready to create a new release. Here we have a choice. If we just run mix release we'll create a whole new releasable application. To deploy it, we'd basically copy it just as we did before, then stop the old app and start the new one.

The alternative is to deploy an *upgrade*

release (sometimes called a *hot upgrade*). This is code that will upgrade the application while it is still running—there should be no downtime. This is one reason why Elixir apps can achieve such high availability numbers. Let's take the upgrade path:

The key thing to note is the creation of the .appup file. This is what tells the Erlang runtime how to upgrade our running app.

Deploying an Upgrade

The deployment of the first release of an app is special: it has to create an environment for that app. With that in place, this release (and all subsequent

releases) will be slightly different. We have to create a release directory on the server and copy the tarball into it. The directory will be under deploy/releases, and will be named the same as the release's version number.

Now let's upgrade the running code:

Head back over to the terminal session that's talking to the app. Don't restart it —just make another request:

Erlang can actually run two versions of a module at the same time. Currently executing code will continue to use the

old version until that code explicitly cites the name of the module that has changed. At that point, and for that particular process, execution will swap to the new version.

This is a critical part of hot loading of code. We want to let code that is currently running continue without interruption, but the new release may not be compatible with it. So Erlang lets it run on the old release. But the next request will reference the module explicitly, and the new code will be loaded.

Here, when we say Sequence.Server.next_number, the reference to Sequence.Server triggers the reload, so the 0.2.0 release handles the

next request.

What if our new release was a disaster? That's not a problem—we can always downgrade to a previous version.

Cool. Let's go back to the current version before continuing.

Migrating Server State

Our boss calls. We're about to go for a second round of funding for our wildly successful sequence-server business, but customers have noticed a bug. We implemented increment_number to add a delta to the current number—a one-time

change. But apparently it was instead supposed to set the difference between successive numbers we served.

Let's try the existing code in our already-running console:

Yup, we're applying the delta only once.

Well, that's an easy change to the code. We simply have to keep one extra thing in the state—a delta value. Here's the updated server code:

otp-app/sequence_v2/lib/sequence/server.ex

The big change is that we made the state a struct rather than a tuple and added the

delta value. We updated the increment handler to change the value of delta, and the next number handler now adds in the delta each time.

The format of the state changed, so we updated the version number (@vsn) to 1.

If we simply stop the old server and start the new one, we'll lose the state stored in the old one. But we can't just copy the state across—the old server had a single integer and the new one has a struct.

Fortunately, OTP has a callback for this. In the new server, implement the code_change function.

otp-app/sequence_v2/lib/sequence/server.ex

The callback takes three arguments—the old version number, the old state, and an additional parameter we don't use. The callback's job is to return {:ok, *new_state*}. In our case, the new state is a struct containing the stash PID and the old current number, along with the new delta value, initialized to 1. We'll need to bump the version number in mix.exs.

otp-app/sequence_v2/mix.exs

Time to create the new release:

Copy it into the deployment location:

Cross your fingers, and upgrade the app:

But the real magic happened over in the console window:

That's the logging we added to our code_change function. We seem to have migrated the server's state into our new structure. Let's try it out:

That's the new behavior, running with our new state structure. We updated the code twice and migrated data once, all while the application continued to run. There was no service disruption, and no loss of data.

Plutarch records the story of a ship

called *Theseus*. Over the course of many years most of the ship's structure was replaced, piece by piece. While this was happening, the ship was in continuous use. Plutarch raises the question, "Is the renovated *Theseus* the same as the original?"

Using Elixir release management, our applications can work the same way the *Theseus* did, running continuously but being updated all the time.

Is the latest application the same as the original? Who cares, as long as it's still running?

OTP Is Big—Unbelievably Big

This book barely scratches OTP's surface. But (I hope) it does introduce the major concepts and give you an idea of what's possible.

More advanced uses of OTP may include release management (including hot code-swapping), handling of distributed failover, automated scaling, and so on. But if you have an application that needs such things, you likely will already have or will soon need dedicated operations experts who know the low-level details of making OTP apps perform the way you need them to.

There is never anything simple about scaling out to the kind of size and sophistication that is possible with OTP. But now you know you can start small and get there.

However, there are ways of writing *some* OTP servers more simply, and that's the subject of the next chapter.

Your Turn

- Exercise: OTP-Applications-3

  Our boss notices that after we applied our version-0-to-version-1 code change, the delta indeed works as specified. However, she also notices that if the server crashes, the delta is forgotten—only the current number is retained. Create a new release that stashes both values.

Copyright © 2018, The Pragmatic Bookshelf.

Chapter 21

Tasks and Agents

This part of the book is about processes and process distribution. So far we've covered two extremes. In the first chapters we looked at the spawn primitive, along with message sending and receiving and multinode operations. We then looked at OTP, the 800-pound gorilla of process architecture.

Sometimes, though, we want something in the middle. We want to be able to run simple processes, either for background processing or for maintaining state. But we don't want to be bothered with the

low-level details of spawn, send, and receive, and we really don't need the extra control that writing our own GenServer gives us.

Enter tasks and agents, two simple-to-use Elixir abstractions. These use OTP's features but insulate us from these details.

(worker)

Tasks

An Elixir task is a function that runs in the background.

tasks/tasks1.exs

The call to Task.async creates a separate process that runs the given function. The return value of async is a task descriptor (actually a PID and a ref) that we'll use to identify the task later.

Once the task is running, the code continues with other work. When it wants to get the function's value, it calls Task.await, passing in the task descriptor. This call waits for our

background task to finish and returns its value.

When we run this, we see

We can also pass Task.async the name of a module and function, along with any arguments. Here are the changes:

tasks/tasks2.exs

Tasks and Supervision

Tasks are implemented as OTP servers, which means we can add them to our application's supervision tree. We can do this in a number of ways.

First, we can link a task to a currently

supervised process by calling start_link instead of async. This has less impact than you might think. If the function running in the task crashes and we use start_link, our process will be terminated immediately. If instead we use async, our process will be terminated only when we subsequently call await on the crashed task.

The second way to supervise tasks is to run them directly from a supervisor. Here we specify the Task module itself as the module to run, and pass it the function to be run in the background as a parameter.

You can take this approach a step

further by moving the task's code out of the supervisor and into its own module.

tasks/my_app/lib/my_app/my_task.ex

The key thing here is use Task. This defines a child_spec function, allowing this module to be supervised:

The problem with this approach is that you can't use Task.await, because your code is not directly calling Task.async.

The solution to this is to supervise the tasks dynamically. This is similar in concept to using a :simple_one_for_one supervisor strategy for regular servers. See the Task documentation for details. [37]

However, before you get too carried away, remember that a simple start_link in an already-supervised process may well be all you need.

Agents

An agent is a background process that maintains state. This state can be accessed at different places within a process or node, or across multiple nodes.

The initial state is set by a function we pass in when we start the agent.

We can interrogate the state using Agent.get, passing it the agent descriptor and a function. The agent runs the function on its current state and returns the result.

We can also use Agent.update to change the state held by an agent. As with the

get operator, we pass in a function. Unlike with get, the function's result becomes the new state.

Here's a bare-bones example. We start an agent whose state is the integer 0. We then use the identity function, &(&1), to return that state. Calling Agent.update with &(&1 + 1) increments the state, as verified by a subsequent get.

In the previous example, the variable count holds the agent process's PID. We can also give agents a local or global name and access them using this name. In this case we exploit the fact that an uppercase bareword in Elixir is converted into an atom with the prefix

Elixir., so when we say Sum it is actually the atom :Elixir.Sum.

The following example shows a more typical use. The Frequency module maintains a list of word/frequency pairs in a map. The dictionary itself is stored in an agent, which is named after the module.

This is all initialized with the start_link function, which, presumably, is invoked during application initialization.

tasks/agent_dict.exs

We can play with this code in IEx.

In a way, you can look at our Frequency module as the implementation part of a gen_server—using agents has simply abstracted away all the housekeeping we had to do.

A Bigger Example

Let's rewrite our anagram code to use both tasks and an agent.

We'll load words in parallel from a number of separate dictionaries. A separate task handles each dictionary. We'll use an agent to store the resulting list of words and signatures.

tasks/anagrams.exs

Our four wordlist files contain the following:

Let’s run it:

Making It Distributed

Agents and tasks run as OTP servers, so they are easy to distribute—just give our agent a globally accessible name. That’s a one-line change:

Now we’ll load our code into two separate nodes and connect them. (Remember that we have to specify names for the nodes so they can talk.)

We’ll start the dictionary agent in node one—this is where the actual dictionary will end up. We’ll then load the

dictionary using both nodes one and two:

Finally, we'll query the agent from both nodes:

Agents and Tasks, or GenServer?

When do you use agents and tasks, and when do you use a GenServer?

The answer is to use the simplest approach that works. Agents and tasks are great when you're dealing with very specific background activities, whereas GenServers (as their name suggests) are more general.

You can eliminate the need to make a decision by wrapping your agents and tasks in modules, as we did in our anagram example. That way you can always switch from the agent or task implementation to the full-blown

GenServer without affecting the rest of the code base.

It’s time to move on and look at some advanced Elixir.

Footnotes

[37] https://hexdocs.pm/elixir/Task.html

Copyright © 2018, The Pragmatic Bookshelf.

Part 3
More Advanced Elixir

Among the joys of Elixir is that it laughs at the concept of "what you see is what you get." Instead, you can extend it in many different ways. This allows you to add layers of abstraction to your code, which makes your code easier to work with.

This part covers macros (which let you extend the language's syntax), protocols (which let you add behaviors to existing

modules), and use (which lets you add capabilities to a module). We finish with a grab-bag chapter of miscellaneous Elixir tricks and tips.

Chapter 22

Macros and Code Evaluation

Have you ever felt frustrated that a language didn't have just the right feature for some code you were writing? Or have you found yourself repeating chunks of code that weren't amenable to factoring into functions? Or have you just wished you could program closer to your problem domain?

If so, then you'll love this chapter.

But, before we get into the details, here's a warning: macros can easily

make your code harder to understand, because you're essentially rewriting parts of the language. For that reason, never use a macro when you could use a function. Let's repeat that:

In fact, you'll probably not write a macro in regular application code. But if you're writing a library and want to use some of the metaprogramming techniques that we show in later chapters, you'll need to know how macros work.

Implementing an if Statement

Let's imagine that Elixir didn't have an if statement—that all it has is case. Although we're prepared to abandon our old friend the while loop, not having an if statement is just too much to bear, so we set about implementing one.

We'll want to call it using something like

We know that blocks in Elixir are converted into keyword parameters, so this is equivalent to

Here's a sample call:

Let's try to implement myif as a function:

When we run it, we're (mildly) surprised to get the following output:

When we call the myif function, Elixir has to evaluate all of its parameters before passing them in. So both the do: and else: clauses are evaluated, and we see their output. Because IO.puts returns :ok on success, what actually gets passed to myif is

This is why the final return value is :ok.

We need a way of delaying the execution of these clauses. This is where macros come in. But before we implement our myif macro, we need a little background.

Macros Inject Code

Let's pretend we're the Elixir compiler. We read a module's source top to bottom and generate a representation of the code we find. That representation is a nested Elixir tuple.

If we want to support macros, we need a way to tell the compiler that we'd like to manipulate a part of that tuple. We do that using defmacro, quote, and unquote.

In the same way that def defines a function, defmacro defines a macro. You'll see what that looks like shortly. However, the real magic starts not when

we define a macro, but when we use one.

When we pass parameters to a macro, Elixir doesn't evaluate them. Instead, it passes them as tuples representing their code. We can examine this behavior using a simple macro definition that prints out its parameter.

macros/dumper.exs

This shows us that atoms, numbers, lists (including keyword lists), binaries, and tuples with two elements are represented internally as themselves. All other Elixir code is represented by a three-element tuple. Right now, the internals of that representation aren't important.

Load Order

You may be wondering about the structure of the preceding code. We put the macro definition in one module, and the usage of that macro in another. And that second module included a require call.

Macros are expanded before a program executes, so the macro defined in one module must be available as Elixir is compiling another module that uses those macros. The require function tells Elixir to ensure the named module is compiled before the current one. In practice it is used to make the macros defined in one module available in another.

But the reason for the two modules is

less clear. It has to do with the fact that Elixir first compiles source files and then runs them.

If we have one module per source file and we reference a module in file A from file B, Elixir will load the module from A, and everything just works. But if we have a module and the code that uses it in the same file, and the module is defined in the same scope in which we use it, Elixir will not know to load the module's code. We'll get this error:

By placing the code that uses the module My in a separate module, we force My to load.

The quote Function

We've seen that when we pass parameters to a macro they are not evaluated. The language comes with a function, quote, that also forces code to remain in its unevaluated form. quote takes a block and returns the internal representation of that block. We can play with it in IEx:

There's another way to think about quote. When we write "abc", we create a binary containing a string. The double quotes say, "interpret what follows as a string of characters and return the appropriate representation."

quote is the same: it says, "interpret the content of the block that follows as

code, and return the internal representation.”

Using the Representation as Code

When we extract the internal representation of some code (either via a macro parameter or using quote), we stop Elixir from adding it automatically to the tuples of code it is building during compilation—we've effectively created a free-standing island of code. How do we inject that code back into our program's internal representation?

There are two ways.

The first is our old friend the macro. Just like with a function, the value a macro returns is the last expression evaluated in that macro. That expression is

expected to be a fragment of code in Elixir's internal representation. But Elixir does not return this representation to the code that invoked the macro. Instead it injects the code back into the internal representation of our program and returns to the caller the result of *executing* that code. But that execution takes place only if needed.

We can demonstrate this in two steps. First, here's a macro that simply returns its parameter (after printing it out). The code we give it when we invoke the macro is passed as an internal representation, and when the macro returns that code, that representation is injected back into the compile tree.

macros/eg.exs

When we run this, we see

Now we'll change that file to return a different piece of code. We use quote to generate the internal form:

macros/eg1.exs

This generates

Even though we passed IO.puts("hello") as a parameter, it was never executed by Elixir. Instead, it ran the code fragment we returned using quote.

Before we can write our version of if,

we need one more trick—the ability to substitute existing code into a quoted block. There are two ways of doing this: by using the unquote function and with bindings.

The unquote Function

Let's get two things out of the way. First, we can use unquote only inside a quote block. Second, unquote is a silly name. It should really be something like inject_code_fragment.

Let's see why we need this. Here's a simple macro that tries to output the result of evaluating the code we pass it:

Unfortunately, when we run it, it reports

an error:

Inside the quote block, Elixir is just parsing regular code, so the name code is inserted literally into the code fragment it returns. But we don't want that. We want Elixir to insert the evaluation of the code we pass in. And that's where we use unquote. It temporarily turns off quoting and simply injects a code fragment into the sequence of code being returned by quote.

Inside the quote block, Elixir is busy parsing the code and generating its internal representation. But when it hits

the unquote, it stops parsing and simply copies the code parameter into the generated code. After unquote, it goes back to regular parsing.

There's another way of thinking about this. Using unquote inside a quote is a way of deferring the execution of the unquoted code. It doesn't run when the quote block is parsed. Instead it runs when the code generated by the quote block is executed.

Or, we can think in terms of our quote-as-string-literal analogy. In this case, we can make a (slightly tenuous) case that unquote is a little like the interpolation we can do in strings. When we write "sum = #{1 + 2}", Elixir evaluates 1 + 2 and interpolates the result into the

quoted string. When we write `quote do: def unquote(name) do end`, Elixir interpolates the contents of `name` into the code representation it is building as part of the list.

Expanding a List—unquote_splicing

Consider this code:

The list `[3,4]` is inserted, as a list, into the overall quoted list, resulting in `[1,2,[3,4]]`.

If we instead wanted to insert just the elements of the list, we could use `unquote_splicing`.

Remembering that single-quoted strings are lists of characters, this means we can write

Back to Our myif Macro

We now have everything we need to implement an if macro.

macros/myif.ex

It’s worth studying this code.

The if macro receives a condition and a keyword list. The condition and any entries in the keyword list are passed as code fragments.

The macro extracts the do: and/or else: clauses from that list. It is then ready to generate the code for our if statement, so it opens a quote block. That block contains an Elixir case expression. This case expression has to evaluate the condition that is passed in, so it uses unquote to inject that condition's code as its parameter.

When Elixir executes this case statement, it evaluates the condition. At that point, case will match the first clause if the result is nil or false; otherwise it matches the second clause. When a clause matches (and only then), we want to execute the code that was passed in either the do: or else: values in the keyword list, so we use unquote

again to inject that code into the case.

Your Turn

- Exercise: MacrosAndCodeEvaluation-1

  Write a macro called myunless that implements the standard unless functionality. You're allowed to use the regular if expression in it.

- Exercise: MacrosAndCodeEvaluation-2

  Write a macro called times_n that takes a single numeric argument. It should define a function called times_*n* in the caller's module that itself takes a single argument, and that multiplies that argument by *n*. So, calling times_n(3) should create a function called times_3, and calling times_3(4) should return 12. Here's an example of it in use:

Using Bindings to Inject Values

Remember that there are two ways of injecting values into quoted blocks. One is unquote. The other is to use a binding. However, the two have different uses and different semantics.

A binding is simply a keyword list of variable names and their values. When we pass a binding to quote, the variables are set inside the body of that quote.

This is useful because macros are executed at compile time. This means they don't have access to values that are calculated at runtime.

Here's an example. The intent is to have

a macro that defines a function that returns its own name:

We try this out using something like mydef(:some_name). Sure enough, that defines a function that, when called, returns :some_name.

Buoyed by our success, we try something more ambitious:

macros/macro_no_binding.exs

And we're rewarded with this:

At the time the macro is called, the each loop hasn't yet executed, so we have no

valid name to pass it. This is where bindings come in:

macros/macro_binding.exs

Two things happen here. First, the binding makes the current value of name available inside the body of the quoted block. Second, the presence of the bind_quoted: option automatically defers the execution of the unquote calls in the body. This way, the methods are defined at runtime.

As its name implies, bind_quoted takes a quoted code fragment. Simple things such as tuples are the same as normal and quoted code, but for most values you probably want to quote them or use

Macro.escape to ensure that your code fragment will be interpreted correctly.

[illegible]

Macros Are Hygienic

It is tempting to think of macros as some kind of textual substitution—a macro's body is expanded as text and then compiled at the point of call. But that's not the case. Consider this example:

macros/hygiene.ex

Here's the result of running that code:

If the macro body were just substituted in at the point of call, both it and the module Test would share the same scope, and the macro would overwrite

the variable local, so we'd see

But that isn't what happens. Instead, the macro definition has both its own scope and a scope during execution of the quoted macro body. Both are distinct from the scope within the Test module. The upshot is that macros will not clobber each other's variables or the variables of modules and functions that use them.

The import and alias functions are also locally scoped. See the documentation for quote for a full description. This also describes how to turn off hygiene for variables and how to control the stack trace's format if things go wrong while

executing a macro.

[illegible]oted

Other Ways to Run Code Fragments

We can use the function Code.eval_quoted to evaluate code fragments, such as those returned by quote.

By default, the quoted fragment is hygienic, and so does not have access to variables outside its scope. Using var!(:name), we can disable this feature and allow a quoted block to access variables in the containing scope. In this case, we pass the binding to eval_quoted as a keyword list.

Code.string_to_quoted converts a string containing code to its quoted form, and Macro.to_string converts a code fragment back into a string.

We can also evaluate a string directly using Code.eval_string.

Macros and Operators

(This is definitely dangerous ground.)

We can override the unary and binary operators in Elixir using macros. To do so, we need to remove any existing definition first.

For example, the operator `+` (which adds two numbers) is defined in the Kernel module. To remove the Kernel definition and substitute our own, we'd need to do something like the following (which redefines addition to concatenate the string representation of the left and right arguments).

macros/operators.ex

Note that the macro's definition is lexically scoped—the `+` operator is overridden from the point when we import the `Operators` module through the end of the module that imports it. We could also have done the import inside a single method, and the scoping would be just that method.

Digging Deeper

The Code and Macro modules contain the functions that manipulate the internal representation of code.

Check the source of the Kernel module for a list of the majority of the operator macros, along with macros for things such as def, defmodule, alias, and so on. If we look at the source code, we'll see the calling sequence for these. However, many of the bodies will be absent, as the macros are defined within the Elixir source.

eval_quoted

Digging Ridiculously Deep

Here's the internal representation of a simple expression:

It's just a three-element tuple. In this particular case, the first element is the function (or macro), the second is housekeeping metadata, and the third is the arguments.

We know we can evaluate this code fragment using eval_quoted, and we can save typing by leaving off the metadata:

And now we can start to see the promise

(and danger) of a homoiconic language (a language in which the internal representation is expressed in the language itself). Because code is just tuples and because we can manipulate those tuples, we have the ability to rewrite the definitions of existing functions. We can create new code on the fly, and we can do it in a safe way because we can control the scope of both the changes and the access to variables.

Next we'll look at *protocols*, a way of adding functionality to built-in code and of integrating our code into other people's modules.

Your Turn

- Exercise: MacrosAndCodeEvaluation-3

  The Elixir test framework, ExUnit, uses some clever code-quoting tricks. For example, if you assert

  ```
  ```

  you'll get the error

  ```
  ```

  The test code parsed the assertion parameter into the left-hand side, the operator, and the right-hand side.

  The Elixir source code is on GitHub (at https://github.com/elixir-lang/elixir). The implementation of this is in the file elixir/lib/ex_unit/lib/ex_unit/assertions.ex. Spend some time reading this file, and work out how it implements this trick.

  (Hard) Once you've done that, see if you can use the same technique to implement a function that takes an arbitrary arithmetic

expression and returns a natural-language version.

Copyright © 2018, The Pragmatic Bookshelf.

Chapter 23

Linking Modules: Behavio(u)rs and use

When we wrote our OTP server, we wrote a module that started with the code

In this chapter we'll explore what lines such as use GenServer actually do, and how we can write modules that extend the capabilities of other modules that use them.

Behaviours

An Elixir behaviour is nothing more than a list of functions. A module that declares that it implements a particular behaviour must implement all of the associated functions. If it doesn't, Elixir will generate a compilation warning. You can think of a behaviour definition as being like an abstract base class in some object-oriented languages.

A behaviour is therefore a little like an *interface* in Java. A module uses it to declare that it implements a particular interface. For example, an OTP GenServer should implement a standard set of callbacks (handle_call,

handle_cast, and so on). By declaring that our module implements that behaviour, we let the compiler validate that we have actually supplied the necessary interface. This reduces the chance of an unexpected runtime error.

Defining Behaviours

We define a behaviour with @callback definitions.

For example, the mix utility can fetch dependencies from various source-code control systems. Out of the box, it supports git and the local filesystem. However, the interface to the source-code control system (which mix abbreviates internally as *SCM*) is defined using a behaviour, allowing new version-control systems to be added

cleanly.

The behaviour is defined in the module Mix.Scm:

This module defines the interface that modules implementing the behaviour must support. It uses @callback to define the functions in the behaviour. But the syntax looks a little different. That's because we're using a minilanguage: Erlang type specifications. For example, the fetchable? function takes no parameters and returns a Boolean. The format function takes a parameter of type opts (which is defined near the top of the code to be a keyword list) and returns a

string. There's more information on these type specifications here.

In addition to the type specification, we can include module- and function-level documentation with our behaviour definitions.

Declaring Behaviours

Now that we've defined the behaviour, we can declare that another module implements it using the @behaviour attribute. Here's the start of the Git implementation for mix:

The module defines each of the functions declared as callbacks in Mix.SCM. This module will compile

cleanly. However, imagine we'd misspelled fetchable:

When we compile the module, we'd get this error:

Behaviours give us a way of both documenting and enforcing the public functions that a module should implement.

Taking It Further

In the implementation of Mix.SCM for Git, we created a bunch of functions that implemented the behaviour. But those are unlikely to be the only functions in this module. And, unless you're

intimately familiar with the Mix.SCM behaviour, you won't be able to tell the callback functions from the rest.

To remedy this, you can flag the callback functions with the @impl attribute. This takes a parameter: either true or the name of a behaviour (guess which one I prefer).

use and __using__

In one sense, use is a trivial function. You pass it a module along with an optional argument, and it invokes the function or macro _using_ in that module, passing it the argument.

Yet this simple interface gives you a powerful extension facility. For example, in our unit tests we write use ExUnit.Case and we get the test macro and assertion support. When we write an OTP server, we write use GenServer and we get both a behaviour that documents the gen_server callback and default implementations of those callbacks.

Typically, the _using_ callback will be implemented as a macro, as it will be used to invoke code in the original module.

(2,3)

Putting It Together—Tracing Method Calls

Let's work through a larger example. We want to write a module called Tracer. If we use Tracer in another module, entry and exit tracing will be added to any subsequently defined function. For example, given the following:

use/tracer.ex

we'd get this output:

My approach to writing this kind of code is to start by exploring what we

have to work with, and then to generalize. The goal is to metaprogram as little as possible.

It looks as if we have to override the def macro, which is defined in Kernel. So let's do that and see what gets passed to def when we define a method.

use/tracer1.ex

```
```

This outputs

```
```

The definition part of each method is a three-element tuple. The first element is the name, the second is the line on which it is defined, and the third is a list of the parameters, where each parameter

is itself a tuple.

We also get an error: puts_sum_three is undefined. That's not surprising—we intercepted the def that defined it, and we didn't create the function.

You may be wondering about the form of the macro definition: defmacro def(definition, do: _content)…. The do: in the parameters is not special syntax: it's a pattern match on the block passed as the function body, which is a keyword list.

You may also be wondering if we have affected the built-in Kernel.def macro. The answer is no. We've created another macro, also called def, which is defined in the scope of the Tracer

module. In our Test module we tell Elixir not to import the Kernel version of def but instead to import the version from Tracer. Shortly, we'll make use of the fact that the original Kernel implementation is unaffected.

Let's see if we can define a real function given this information. That turns out to be surprisingly easy. We already have the two arguments passed to def. All we have to do is pass them on.

use/tracer2.ex

When we run this, we see 6, the output from puts_sum_three.

Now it's time to add some tracing.

use/tracer3.ex

Looking good:

Let's package our Tracer module so clients only have to add use Tracer to their own modules. We'll implement the _using_ callback. The tricky part here is differentiating between the two modules: Tracer and the module that uses it.

use/tracer4.ex

Use use

Elixir behaviours are fantastic—they let you easily inject functionality into modules you write. And they're not just for library creators—use them in your own code to cut down on duplication and boilerplate.

Although behaviours let you add to modules that you are writing, you sometimes need to extend the functionality of modules written by others—code that you can't change. Fortunately, Elixir comes with *protocols*, the subject of the next chapter.

Your Turn

- Exercise: LinkingModules-BehaviorsAndUse-1

  In the body of the def macro, there's a quote block that defines the actual method. It contains

  ```
  ```

  Why does the first call to puts have to unquote the values in its interpolation but the second call does not?

- Exercise: LinkingModules-BehaviorsAndUse-2

  The built-in module IO.ANSI defines functions that represent ANSI escape sequences. You can use it to build output than will display (for example) colors and bold, inverse, or underlined text (assuming the terminal supports it).

  ```
  ```

  Explore the module, and use it to colorize our tracing's output.

Why does passing a list of strings to IO.puts work?

- Exercise: LinkingModules-BehavioursAndUse-3

  (Hard) Try adding a method definition with a guard clause to the Test module. You'll find that the tracing no longer works.

  - Find out why.
  - See if you can fix it.

Copyright © 2018, The Pragmatic Bookshelf.

Chapter 24

Protocols—Polymorphic Functions

We have used the inspect function many times in this book. It returns a printable representation of any value as a binary (which is what we hard-core folks call strings).

But stop and think for a minute. Just how can Elixir, which doesn't have objects, know what to call to do the conversion to a binary? You can pass inspect anything, and Elixir somehow makes sense of it.

It *could* be done using guard clauses:

But there's a better way.

Elixir has the concept of *protocols*. A protocol is a little like the behaviours we saw in the previous chapter in that it defines the functions that must be provided to achieve something. But a behaviour is internal to a module—the module implements the behaviour. Protocols are different—you can place a protocol's implementation completely outside the module. This means you can extend modules' functionality without having to add code to them—in fact, you can extend the functionality even if you don't have the modules' source

code.

def Nil.to_string(_) do any
end
end

Defining a Protocol

Protocol definitions are very similar to basic module definitions. They can contain module- and function-level documentation (@moduledoc and @doc), and they will contain one or more function definitions. However, these functions will not have bodies—they are there simply to declare the interface that the protocol requires.

For example, here is the definition of the Inspect protocol:

```
```

Just like a module, the protocol defines one or more functions. But we

implement the code separately.

```
end
end
```

Implementing a Protocol

The defimpl macro lets you give Elixir the implementation of a protocol for one or more types. The code that follows is the implementation of the Inspect protocol for PIDs and references.

Finally, the Kernel module implements inspect, which calls Inspect.inspect with its parameter. This means that when you call inspect(self), it becomes a call to Inspect.inspect(self). And because self is a PID, this in turn resolves to something like "#PID<0.25.0>".

Behind the scenes, defimpl puts the

implementation for each protocol-and-type combination into a separate module. The protocol for Inspect for the PID type is in the module Inspect.PID. And because you can recompile modules, you can change the implementation of functions accessed via protocols.

The Available Types

You can define implementations for one or more of the following types:

The type BitString is used in place of Binary.

The type Any is a catchall, allowing you to match an implementation with any type. Just as with function definitions, you'll want to put the implementations for specific types before an implementation for Any.

You can list multiple types on a single defimpl. For example, the following

protocol can be called to determine whether a type is a collection:

protocols/is_collection.exs

We write defimpl stanzas for the collection types: List, Tuple, BitString, and Map. But what about the other types? To handle those, we use the special type Any in a second defimpl. If we use Any, though, we also have to add an annotation to the protocol definition. That's what the @fallback_to_any line does.

This produces

Your Turn

- Exercise: Protocols-1

  A basic Caesar cypher consists of shifting the letters in a message by a fixed offset. For an offset of 1, for example, a will become b, b will become c, and z will become a. If the offset is 13, we have the ROT13 algorithm.

  Lists and binaries can be *stringlike*. Write a Caesar protocol that applies to both. It would include two functions: encrypt(string, shift) and rot13(string).

- Exercise: Protocols-2

  Using a list of words in your language, write a program to look for words where the result of calling rot13(word) is also a word in the list. (For various English word lists, look at http://wordlist.sourceforge.net/. The SCOWL collection looks promising, as it already has words divided by size.)

Table of Content:

Protocols and Structs

Elixir doesn't have classes, but (perhaps surprisingly) it does have user-defined types. It pulls off this magic using structs and a few conventions.

Let's play with a simple struct. Here's the definition:

protocols/basic.exs

And here we use it in IEx:

It looks for all the world as if we've created some new type, the blob. But that's only because Elixir is hiding

something from us. By default, inspect recognizes structs. If we turn this off using the structs: false option, inspect reveals the true nature of our blob value:

A struct value is actually just a map with the key _struct_ referencing the struct's module (Blob in this case) and the remaining elements containing the keys and values for this instance. The inspect implementation for maps checks for this —if you ask it to inspect a map containing a key _struct_ that references a module, it displays it as a struct.

Many built-in types in Elixir are represented as structs internally. It's

instructive to try creating values and inspecting them with structs: false.

Built-in Protocols

Elixir comes with the following protocols:

- Enumerable and Collectable
- Inspect
- List.Chars
- String.Chars

To play with these, let's work with MIDI files.

A MIDI file consists of a sequence of variable-length frames. Each frame contains a four-character type, a 32-bit length, and then *length* bytes of data.[38]

We'll define a module that represents the MIDI file content as a struct,

because the struct lets us use it with protocols. The file also defines a submodule for the individual frame structure.

protocols/midi.exs

Built-in Protocols: Enumerable and Collectable

The Enumerable protocol is the basis of all the functions in the Enum module—any type implementing it can be used as a collection argument to Enum functions.

We're going to implement Enumerable for our Midi structure, so we'll need to wrap the implementation in something like this:

The protocol is defined in terms of four functions:

`count` returns the number of elements in the collection, `member?` is truthy if the collection contains `value`, and `reduce` applies the given function to successive values in the collection and the accumulator; the value it reduces becomes the next accumulator. Finally, `slice` is used to create a subset of a collection. Perhaps surprisingly, all the `Enum` functions can be defined in terms of these four.

However, life isn't that simple. Maybe you're using `Enum.find` to find a value in a large collection. Once you've found

it, you want to halt the iteration—continuing is pointless. Similarly, you may want to suspend an iteration and resume it sometime later. These two features become particularly important when we talk about streams, which let you enumerate a collection lazily.

We'll start with the most difficult function to implement, Enumerable.reduce/3. It is worth reading the documentation for it before we start:

The first two function heads do housekeeping: they handle the cases where the enumeration has been halted

or suspended. Here are the versions for our MIDI enumerator:

protocols/midi.exs

The next two function heads do the actual iteration. In the list example in the documentation, you'll see the typical pattern: check for the end condition ([ ]) and the recursive step [h|t].

We'll do the same with our MIDI file, but we'll use binaries instead of doing list pattern matching:

protocols/midi.exs

See how we split out the binary content of a frame, then wrap it into a

Midi.Frame struct before passing it back. This means that folks who use our MIDI module will only see these frame structures, and not the raw data.

Before we try this, there's one little tweak we have to make. You may have noticed that all my functions were named _reduce, with a leading underscore. That's because they need to work on the content of the MIDI file, and not on the structure that wraps that content. We have a single function head that implements the actual reduce function, and that then forwards the call on to _reduce:

protocols/midi.exs

At this point we have enough code to try it out. I've included a MIDI file in the code/protocols directory for you to play with (courtesy of midiworld.com).[39]

First, see how we get warnings because we haven't yet implemented the full Enumerable protocol. Normally this would worry me, but I happen to know that we won't be using those functions just yet.

Next, look at how we called Enum.take/2 and got back two Midi.Frame structures. That's because take/2 is defined in terms of reduce/3, and we supplied an implementation of it.

So far, so good. Let's implement `count/1` next.

If the collection is countable, the `count` function returns a tuple containing `{:ok, count}`. If it isn't countable (perhaps it is being read one piece at a time from an external source), `count` should return `{:ok, _MODULE_}`.

In our case, we have the whole MIDI file available in memory, and we have a way to traverse it using `reduce`, so counting it easy:

protocols/midi.exs

Let's try it:

On to member? and slice.

Technically, both of these can be implemented using reduce. But the Elixir team recognized that some types of collection have more direct ways to test membership and partition elements. For example, if you implement a set using a map, then testing to see if a key is present can be done in constant time. If you have an array-like structure with fixed-length elements, you can split it into two in (almost) constant time.

So the implementations of both member? and slice depend on the characteristics of your collection. In our case, we don't currently have a fast way of testing for membership, nor do we have a fast way to slice a MIDI file into

two. In both cases, we return an error tuple. This doesn't actually cause an error for the user; it just tells Enumerable to fall back to a naive algorithm.

protocols/midi.exs

And with that, we're done. Our Midi type is enumerable, and you can use every function in Enum on it.

This gives us the ability to treat a MIDI stream as a collection of MIDI frames. But how do we assemble frames back into a MIDI stream? That's what we'll address next.

Collectable

We've already seen Enum.into/2. It

takes something that's enumerable and creates a new collection from it:

The target of Enum.into must implement the Collectable protocol. This defines a single function, somewhat confusingly also called into. This function returns a two-element tuple. The first element is the initial value of the target collection. The second is a function to be called to add each item to the collection. (If this reminds you of the second and third parameters passed to Enum.reduce, that's because in a way into is the opposite of reduce.)

Let's look at the code first:

protocols/midi.exs

It works like this:

- Enum.into calls the into function for Midi, passing it the target value—Midi{content: content} in this case.
- Midi.into returns a tuple. The first element is the current content of the target. This acts as the initial value for an accumulator. The second element of the tuple is a function.
- Enum.into then calls this function, passing it the accumulator and a command. If the command is :done, the iteration over the collection being injected into the MIDI stream has finished, so we return a new Midi structure using the accumulator as a value. If the command is :halt, the iteration has terminated early and nothing needs to be done.
- The real work is done when the function is passed the {:cont, frame} command. Here is where the Collectable appends the binary representation of the next frame to the accumulator.

We can call it in IEx:

Because the into function uses the initial value of the target collection, we can use it to append to a MIDI stream:

Remember the Big Picture

If you think all this enumerable/collectable stuff is complicated—well, you're correct. It is. In part that's because these conventions allow all enumerable values to be used both eagerly and lazily. And when you're dealing with big (or even infinite) collections, this is a big deal.

Built-in Protocols: Inspect

This is the protocol that is used to

inspect a value. The rule is simple—if you can return a representation that is a valid Elixir literal, do so. Otherwise, prefix the representation with #Typename.

We *could* just delegate the inspect function to the Elixir default. (That's what we've been doing so far.) But we can do better. Not surprisingly, we do that by implementing the Inspect protocol. We'll do it for both the overall Midi type and for the individual Midi.Frames.

protocols/midi_inspect.exs

Run the code in IEx:

I added a little bit of decoding for the header frame, but just treated the track frames as binary. You could extend this to do a full decode of each track frame too.

There's a wrinkle here. If you pass `structs: false` to `IO.inspect` (or `Kernel.inspect`), it never calls our `inspect` function. Instead, it formats it as a struct.

Better Formatting with Algebra Documents

The formatting of our MIDI stream leaves a little to be desired: there's no indentation or reasonable line wrapping.

To fix this, we use a feature called *algebra documents*. An algebra document is a tree structure that

represents some data you'd like to pretty-print.[40] Your job is to create the structure based on the data you want to inspect, and Elixir will then find a nice way to display it.

I'd like the inspect string to show the nesting of data, and to wrap long lines honoring that nesting.

We do this by having our inspect function return an algebra document rather than a string. In that document, we indicate places where breaks are allowed (but not required) and we show how the nesting works:

protocols/midi_algebra.exs

On a narrow terminal window, we get this output:

For more information, see the documentation for Inspect.Algebra.

Built-in Protocols: List.Chars and String.Chars

The List.Chars protocol is used by Kernel.to_charlist to convert a value into a list of characters (think *single-quoted string*).

The String.Chars protocol is used to convert a value to a string (binary, or *double-quoted string*). This is the protocol used for string interpolation.

The protocols are implemented identically, except List.Chars requires

that you write a to_charlist function, and String.Chars requires you write to_string.

Although we *could* implement a String.Chars.to_string for our Midi struct, it probably wouldn't make much sense. What would you interpolate into the string?

Protocols Are Polymorphism

When you want to write a function that behaves differently depending on the type of its arguments, you're looking at a polymorphic function. Elixir protocols give you a tidy and controlled way to implement this. Whether you're integrating your types into the existing Elixir library or creating a new library with a flexible interface, protocols let you package the behaviour in a well-documented and disciplined way. And with that, we're almost done. But when you write about a language, there are always little details that don't seem to fit anywhere. That's why the next chapter is full of odds and ends.

Your Turn

- Exercise: Protocols-3

  Collections that implement the Enumerable protocol define count, member?, reduce, and slice functions. The Enum module uses these to implement methods such as each, filter, and map.

  Implement your own versions of each, filter, and map in terms of reduce.

- Exercise: Protocols-4

  In many cases, inspect will return a valid Elixir literal for the value being inspected. Update the inspect function for structs so that it returns valid Elixir code to construct a new struct equal to the value being inspected.

Footnotes

[38] https://www.csie.ntu.edu.tw/~r92092/ref/midi/

[39] http://midiworld.com

[40]

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.2200

Copyright © 2018, The Pragmatic Bookshelf.

Chapter 25

More Cool Stuff

Elixir is packed with features that make coding a joy. This chapter contains a smattering of them.

Writing Your Own Sigils

You know by now that you can create strings and regular-expression literals using sigils:

Have you ever wished you could extend these sigils to add your own specific literal types? You can.

When you write a sigil such as `~s{...}`, Elixir converts it into a call to the function `sigil_s`. It passes the function two values. The first is the string between the delimiters. The second is a list containing any lowercase letters that immediately follow the closing

delimiter. (This second parameter is used to pick up any options you pass to a regex literal, such as ~r/cat/if.)

Here's the implementation of a sigil ~l that takes a multiline string and returns a list containing each line as a separate string. We know that ~l… is converted into a call to sigil_l, so we just write a simple function in the LineSigil module.

odds/line_sigil.exs

We can play with this in a separate module:

odds/line_sigil.exs

This produces ["line 1","line 2","and another line in Elixir.Example"].

Because we import the sigil_l function inside the example module, the ~l sigil is lexically scoped to this module. Note also that Elixir performs interpolation before passing the string to our method. That's because we used a lowercase l. If our sigil were ~L{…} and the function were renamed sigil_L, no interpolation would be performed.

The predefined sigil functions are sigil_C, sigil_c, sigil_R, sigil_r, sigil_S, sigil_s, sigil_W, and sigil_w. If you want to override one of these, you'll need to explicitly import the Kernel module and use an except clause to exclude it.

In this example, we used the heredoc syntax (""""). This passes our function a multiline string with leading spaces removed. Sigil options are not supported with heredocs, so we'll switch to a regular literal syntax to play with them.

Picking Up the Options

Let's write a sigil that enables us to specify color constants. If we say `~c{red}`, we'll get `0xff0000`, the RGB representation. We'll also support the option `h` to return an HSB value, so `~c{red}h` will be `{0,100,100}`.

Here's the code:

odds/color.exs

The three clauses for the sigil_c function let us select the colorspace to use based on the option passed. As the single-quoted string 'r' is actually represented by the list [?r], we can use the string literal to pattern-match the options parameter.

Because I'm overriding a built-in sigil, I decided to implement a _using_ macro that automatically removes the Kernel version and adds our own (but only in the lexical scope that calls use on our module).

The fact that we can write our own sigils is liberating. But misuse could lead to some pretty impenetrable code.

Your Turn

- Exercise: MoreCoolStuff-1

  Write a sigil `~v` that parses multiple lines of comma-separated data, returning a list where each element is a row of data and each row is a list of values. Don't worry about quoting—just assume each field is separated by a comma.

  For example

  ```
  ```

  would generate `[["1","2","3"], ["cat","dog"]]`.

- Exercise: MoreCoolStuff-2

  The function `Float.parse` converts leading characters of a string to a float, returning either a tuple containing the value and the rest of the string, or the atom `:error`.

  Update your CSV sigil so that numbers are automatically converted:

  ```
  ```

should generate `[[1.0,2.0,3.14], ["cat","dog"]]`.

- Exercise: MoreCoolStuff-3

  (Hard) Sometimes the first line of a CSV file is a list of the column names. Update your code to support this, and return the values in each row as a keyword list, using the column names as the keys. Here's an example:

  ```
  ```

  would generate

  ```
  ```

Multi-app Umbrella Projects

It is unfortunate that Erlang chose to call self-contained bundles of code *apps*. In many ways, they are closer to being shared libraries. And as your projects grow, you may find yourself wanting to split your code into multiple libraries, or apps. Fortunately, mix makes this painless.

To illustrate the process, we'll create a simple Elixir evaluator. Given a set of input lines, it will return the result of evaluating each. This will be one app.

To test it, we'll need to pass in lists of lines. We've already written a trivial ~l

sigil that creates lists of lines for us, so we'll make that sigil code into a separate application.

Elixir calls these multi-app projects *umbrella projects*.

Create an Umbrella Project

We use mix new to create an umbrella project, passing it the --umbrella option.

Compared to a normal mix project, the umbrella is pretty lightweight—just a mix file and an apps directory.

Create the Subprojects

Subprojects are stored in the apps directory. There's nothing special about them—they are simply regular projects

created using mix new. Let's create our two projects now:

At this point we can try out our umbrella project. Go back to the overall project directory and try mix compile.

Now we have an umbrella project containing two regular projects. Because there's nothing special about the subprojects, you can use all the regular mix commands in them. At the top level, though, you can build all the subprojects as a unit.

Making the Subproject Decision

The fact that subprojects are just regular

mix projects means you don't have to worry about whether to start a new project using an umbrella. Simply start as a simple project. If you later discover the need for an umbrella project, create it and move your existing simple project into the apps directory.

The LineSigil Project

This project is trivial—just copy the LineSigil module from the previous section into apps/line_sigil/lib/line_sigil.ex. Verify it builds by running mix compile—in either the top-level directory or the line_sigil directory.

The Evaluator Project

The evaluator takes a list of strings containing Elixir expressions and evaluates them. It returns a list

containing the expressions intermixed with the value of each. For example, given

our code will return

We'll use Code.eval_string to execute the Elixir expressions. To have the values of variables pass from one expression to the next, we'll also need to explicitly maintain the current binding.

Here's the code:

odds/eval/apps/evaluator/lib/evaluator.ex

Linking the Subprojects

Now we need to test our evaluator. It makes sense to use our ~l sigil to create lists of expressions, so let's write our tests that way.

odds/eval/apps/evaluator/test/evaluator_test.exs

But if we simply run this in the apps/evaluator directort, Elixir won't be able to find the LineSigil module, as we don't have a dependency. Instead, just run the tests from the top-level directory (the one containing the overall umbrella project). Mix will automatically load all the child apps for you.

The first stanza of test output is for the evaluator tests, and the second is for

line_sigil.

But Wait! There's More!

We've reached the end of our Elixir exploration.

This book was never intended to be exhaustive. Instead, it is intended to hit the highlights, and to give you enough information to start coding apps in Elixir yourself.

That means there's a lot more to learn, both about the language and about how to write great apps in it.

And I think that's fun. Enjoy!

Copyright © 2018, The Pragmatic Bookshelf.

Appendix 1

Exceptions: raise and try, catch and throw

Elixir (like Erlang) takes the view that errors should normally be fatal to the processes in which they occur. A typical Elixir application's design involves many processes, which means the effects of an error will be localized. A supervisor will detect the failing process, and the restart will be handled at that level.

For that reason, you won't find much exception-handling code in Elixir programs. Exceptions are raised, but

you rarely catch them.

Use exceptions for things that are exceptional—things that should never happen.

Exceptions do exist. This appendix is an overview of how to generate them and how to catch them when they occur.

Raising an Exception

You can raise an exception using the raise function. At its simplest, you pass it a string and it generates an exception of type RuntimeError.

You can also pass the type of the exception, along with other optional fields. All exceptions implement at least the message field.

You can intercept exceptions using the try function. It takes a block of code to execute, and optional rescue, catch, and after clauses.

The rescue and catch clauses look a bit like the body of a case function—they take patterns and code to execute if the pattern matches. The subject of the pattern is the exception that was raised.

Here's an example of exception handling in action. We define a module that has a public function, start. It calls a different helper function depending on the value of its parameter. With 0, it runs smoothly. With 1, 2, or 3, it causes the VM to raise an error, which we catch and report.

exceptions/exception.ex

We define three different exception patterns. The first matches one of the

two exceptions, FunctionClauseError or RuntimeError. The second matches an ArithmeticError and stores the exception value in the variable error. And the last clause catches *any* exception into the variable other_error.

We also include an after clause. This will always run at the end of the try function, regardless of whether an exception was raised.

Finally, look at the handling of ArithmeticError. As well as reporting the error, we call reraise. This raises the current exception, but lets us add a message. We also pass in the stack trace (which is actually the stack trace at the point the original exception was raised). Let's see all this in IEx:

bad_happened

catch, exit, and throw

Elixir code (and the underlying Erlang libraries) can raise a second kind of error. These are generated when a process calls error, exit, or throw. All three take a parameter, which is available to the catch handler.

Here's an example:

exceptions/catch.ex

Calling the start function with 1, 2, or 3 will cause an exit, a throw, or an error to be thrown. Just to illustrate wildcard pattern matching, we handle the last case by matching any type into the

variable what.

Defining Your Own Exceptions

Exceptions in Elixir are basically records. You can define your own exceptions by creating a module. Inside it, use defexception to define the various fields in the exception, along with their default values. Because you're creating a module, you can also add functions—often these are used to format the exception's fields into meaningful messages.

Say we're writing a library to talk to a Microsoft Kinect controller. It might want to raise an exception on various kinds of communication errors. Some of these are permanent, but others are

likely to be transient and can be retried. We'll define our exception with its (required) message field and an additional can_retry field. We'll also add a function that formats these two fields into a nice message.

exceptions/defexception.ex

Users of our library could write code like this:

exceptions/defexception.ex

If an exception gets raised, the code handles it and possibly retries:

Now Ignore This Appendix

The Elixir source code for the mix utility contains no exception handlers. The Elixir compiler itself contains a total of five (but it is doing some pretty funky things).

If you find yourself defining new exceptions, ask if you should be isolating the code in a separate process instead. After all, if it can go wrong, wouldn't you want to isolate it?

Copyright © 2018, The Pragmatic Bookshelf.

Appendix 2

Type Specifications and Type Checking

When we looked at @callback,, we saw that we defined callbacks in terms of their parameter types and return values. For example, we might write

The terms URI.Info.t and integer are examples of type specifications. And, as José Valim pointed out to me, the cool thing is that they are implemented (by Yurii Rashkovskii) directly in the Elixir language itself—no special parsing is

involved. This is a great illustration of the power of Elixir metaprogramming.

In this appendix we'll discuss how to specify types in Elixir. But before we do, there's another question to address: *Why bother?*

integer(),

When Specifications Are Used

Elixir type specifications come from Erlang. It is very common to see Erlang code where every exported (public) function is preceded by a -spec line. This is metadata that gives type information. The following code comes from the Elixir parser (which is [currently] written in Erlang). It says the return_error function takes two parameters, an integer and any type, and never returns.

One of the reasons the Erlang folks do this is to document their code. You can read it inline while reading the source,

and you can also read it in the pages created by their documentation tool.

The other reason is that they have tools such as *dialyzer* that perform static analysis of Erlang code and report on some kinds of type mismatches.[41]

These same benefits can apply to Elixir code. We have the `@spec` module attribute for documenting a function's type specification; in IEx we have the `s` helper for displaying specifications and the `t` helper for showing user-defined types. You can also run Erlang tools such as dialyzer on compiled Elixir `.beam` files.

However, type specifications are not currently in wide use in the Elixir world.

Whether you use them is a matter of personal taste.

Specifying a Type

A type is simply a subset of all possible values in a language. For example, the type integer means all the possible integer values, but excludes lists, binaries, PIDs, and so on.

The basic types in Elixir are as follows: any, atom, float, fun, integer, list, map, maybe_improper_list, none, pid, port, reference, struct, and tuple.

The type any (and its alias, _) is the set of all values, and none is the empty set.

A literal atom or integer is the set containing just that value.

The value nil can be represented as nil.

Collection Types

A list is represented as [*type*], where *type* is any of the basic or combined types. This notation does not signify a list of one element—it simply says that elements of the list will be of the given type. If you want to specify a nonempty list, use [*type*, ...]. As a convenience, the type list is an alias for [any].

Binaries are represented using this syntax:

<< >>

An empty binary (size 0).

<< _ :: *size* >>

A sequence of *size* bits. This is

called a *bitstring*.

`<< _ :: size * unit_size >>`

A sequence of *size* units, where each unit is *unit_size* bits long.

In the last two instances, *size* can be specified as _, in which case the binary has an arbitrary number of bits/units.

The predefined type bitstring is equivalent to `<<_::_>>`, an arbitrarily sized sequence of bits. Similarly, binary is defined as `<<_::_*8>>`, an arbitrary sequence of 8-bit bytes.

Tuples are represented as { *type, type,…* } or using the type tuple, so both {atom,integer} and tuple(atom,integer} represent a tuple whose first element is

an atom and whose second element is an integer.

Combining Types

The range operator (`..`) can be used with literal integers to create a type representing that range. The three built-in types, `non_neg_integer`, `pos_integer`, and `neg_integer`, represent integers that are greater than or equal to, greater than, or less than zero, respectively.

The union operator (`|`) indicates that the acceptable values are the unions of its arguments.

Parentheses may be used to group terms in a type specification.

Structures

As structures are basically maps, you could just use the map type for them, but doing so throws away a lot of useful information. Instead, I recommend that you define a specific type for each struct:

You can then reference this type as LineItem.t.

Anonymous Functions

Anonymous functions are specified using (*head -> return_type*).

The *head* specifies the arity and possibly the types of the function parameters. Use ". . ." to mean an arbitrary number of arbitrarily typed arguments, or a list

of types, in which case the number of types is the function's arity.

You can put parentheses around the head if you find it clearer:

Handling Truthy Values

The type as_boolean(T) says that the actual value matched will be of type T, but the function that uses the value will treat it as a *truthy* value (anything other than nil or false is considered true). Thus the specification for the Elixir function Enum.count is

Some Examples

integer | float

Any number (Elixir has an alias for this).

[{atom, any}]
list(atom, any)

A list of key/value pairs. The two forms are the same.

non_neg_integer | {:error, String.t}

An integer greater than or equal to zero, or a tuple containing the atom :error and a string.

(integer, atom -> { :pair, atom, integer })

An anonymous function that takes an integer and an atom and returns a tuple containing the atom :pair, an atom, and an integer.

<< _ :: _ * 4 >>

A sequence of 4-bit nibbles.

Defining New Types

The attribute @type can be used to define new types.

Elixir uses this to predefine some built-in types and aliases. Here are just some of them.

As the list(t) entry shows, you can parameterize the types in a new definition. Simply use one or more identifiers as parameters on the left side, and use these identifiers where you'd otherwise use type names on the right. Then when you use the newly defined

type, pass in actual types for each of these parameters:

As well as @type, Elixir has the @typep and @opaque module attributes. They have the same syntax as @type, and do basically the same thing. The difference is in the visibility of the result.

@typep defines a type that is local to the module that contains it—the type is private. @opaque defines a type whose name may be known outside the module but whose definition is not.

integer

Specs for Functions and Callbacks

The @spec specifies a function's parameter count, types, and return-value type. It can appear anywhere in a module that defines the function, but by convention it sits immediately before the function definition, following any function documentation.

We've already seen the syntax:

@spec *function_name*(*param1_type*, …) :: *return_type*

Let's see some examples. These come from the built-in Dict module.

Line 6

values takes a collection (tuple or list) and returns a list of values (any).

Line 7

size takes a collection and returns an integer (>= 0).

Line 8

has_key? takes a collection and a key, and returns true or false.

Line 9

update takes a collection, a key, a value, and a function that maps a value to a value. It returns a (new) collection.

For functions with multiple heads (or

those that have default values), you can specify multiple @spec attributes. Here's an example from the Enum module:

The Enum module also has many examples of the use of as_boolean:

This says filter takes something enumerable and a function. That function maps an element to a term (which is an alias for any), and the filter function treats that value as being truthy. filter returns a list.

For more information on Elixir support for type specifications, see the guide.[42]

[illegible](list)

Using Dialyzer

Dialyzer analyzes code that runs on the Erlang VM, looking for potential errors. To use it with Elixir, we have to compile our source into .beam files and make sure that the debug_info compiler option is set (which it is when running mix in the default, development mode). Let's see how to do that by creating a trivial project with two source files.

Inside the project, let's create a simple function. Being lazy, I haven't implemented the body yet.

Let's run dialyzer on our code. To make life simple, we'll use the dialyxir library to add a dialyzer task to mix:

typespecs/simple/mix.exs

Fetch the library and build our project:

Now we're ready to analyze our code. However, the first time we do this, dialyzer needs to construct a massive data structure containing all the types and APIs in both Erlang and Elixir. This lets it check not just our code, but also that our code is interacting correctly with the rest of the world. Building this data structure is slow: expect it to take 10 to 20 minutes! But once done, it

won't be repeated.

Ouch! Let's run it again:

Those last three lines are the important ones. They're complaining that the typespec for count_atoms doesn't agree with the implementation. The *success typing* (think of this as the *actual type*) [43] returns nil, but the spec says it is a nonnegative integer. Dialyzer has caught our stubbed-out body.

Let's fix that:

typespecs/simple/lib/simple.ex

and run dialyzer again:

Let's add a second module that calls our count_atoms function:

typespecs/simple/lib/simple/client.ex

Compile and dialyze:

That's pretty cool. Dialyzer noticed that we called count_atoms with a list of integers, but it is specified to receive a list of atoms. It also decided this would raise an error, so the function would never return (that's the *no local return* warning). Let's fix that:

And so it goes….

Dialyzer and Type Inference

In this appendix, we've shown dialyzer working with type specs that we added to our functions. But it also does a credible job with unannotated code. This is because dialyzer knows the types of the built-in functions (remember the long wait the first time we ran it) and can infer (some of) your function types from this. Here's a simple example:

Compile this, and run dialyzer:

Here it noticed that the length_plus_n

function called length on its first parameter, and length requires a list as an argument. This means length_plus_n also needs a list argument, and so it complains.

What happens if we change the call to length_plus_n([:a, :b], :c)?

This is even cooler. It knows that + (which is implemented as a function) takes two numeric arguments. When we pass an atom as the second parameter, dialyzer recognizes that this makes no sense, and complains. But look at the error. It isn't complaining about the addition. Instead, it has assigned a

default typespec to our function, based on its analysis of what we call inside that function.

This is *success typing*. Dialyzer attempts to infer the most permissive types that are compatible with the code—it assumes the code is correct until it finds a contradiction. This makes it a powerful tool, as it can make assumptions as it runs.

Does that mean you don't need @spec attributes? That's your call. Try it with and without. Often, adding a @spec will further constrain a function's type signature. We saw this with our count_of_atoms function, where the spec made it explicit that we expected a list of atoms as an argument.

Ultimately, dialyzer is a tool, not a test of your coding chops. Use it as such, but don’t waste time adding specs to get a gold star.

Footnotes

[41] http://www.erlang.org/doc/man/dialyzer.html

[42] http://elixir-lang.org/getting-started/typespecs-and-behaviours.html#types-and-specs

[43] http://www.it.uu.se/research/group/hipe/papers/succ_types.pdf

Copyright © 2018, The Pragmatic Bookshelf.

Bibliography

[Arm13] Joe Armstrong. ***Programming Erlang (2nd edition)***. The Pragmatic Bookshelf, Raleigh, NC, 2nd, 2013.

Copyright © 2018, The Pragmatic Bookshelf.

Thank you!

How did you enjoy this book? Please let us know. Take a moment and email us at support@pragprog.com with your feedback. Tell us your story and you could win free ebooks. Please use the subject line “Book Feedback.”

Ready for your next great

Pragmatic Bookshelf book? Come on over to https://pragprog.com and use the coupon code BUYANOTHER2018 to save 30% on your next ebook.

Void where prohibited, restricted, or otherwise unwelcome. Do not use ebooks near water. If rash persists, see a doctor. Doesn't apply to *The Pragmatic Programmer* ebook because it's older than the Pragmatic Bookshelf itself. Side effects may include increased knowledge and skill, increased marketability, and deep satisfaction. Increase dosage regularly.

And thank you for your continued

support,

Andy Hunt, Publisher



You May Be Interested In…

Select a cover for more information

Programming Phoenix ≥ 1.4



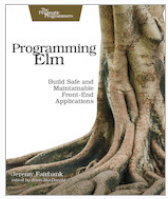
Don’t accept the compromise between fast and beautiful: you can have it all. Phoenix creator Chris McCord, Elixir creator José Valim, and award-winning author Bruce Tate walk you through building an application that’s fast and reliable. At every step, you’ll learn from the Phoenix creators not just what to do, but why. Packed

with insider insights and completely updated for Phoenix 1.4, this definitive guide will be your constant companion in your journey from Phoenix novice to expert, as you build the next generation of web applications.

Chris McCord, Bruce Tate and José Valim

(325 pages) ISBN: 9781680502268 $45.95

Programming Elm



Elm brings the safety and stability of functional programing to front-end development, making it one of the most popular new languages. Elm's functional nature and static typing means that run-time errors are nearly impossible, and it compiles to JavaScript for easy web deployment. This book helps you take advantage of this new language in your web site development. Learn how the Elm Architecture will help you create fast applications. Discover how to integrate Elm with JavaScript so you can update legacy applications. See

how Elm tooling makes deployment quicker and easier.

Jeremy Fairbank

(250 pages) ISBN: 9781680502855 $40.95

Functional Web Development with Elixir, OTP, and Phoenix



Elixir and Phoenix are generating tremendous excitement as an unbeatable platform for building modern web applications. For decades OTP has helped developers create incredibly robust,

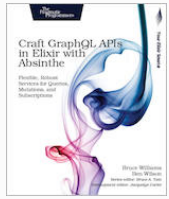
scalable applications with unparalleled uptime. Make the most of them as you build a stateful web app with Elixir, OTP, and Phoenix. Model domain entities without an ORM or a database. Manage server state and keep your code clean with OTP Behaviours. Layer on a Phoenix web interface without coupling it to the business logic. Open doors to powerful new techniques that will get you thinking about web development in fundamentally new ways.

Lance Halvorsen

(218 pages) ISBN:

9781680502435 $45.95

Craft GraphQL APIs in Elixir with Absinthe



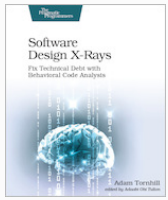
Your domain is rich and interconnected, and your API should be too. Upgrade your web API to GraphQL, leveraging its flexible queries to empower your users, and its declarative structure to simplify your code. Absinthe is the GraphQL toolkit for Elixir, a functional programming language designed to enable massive concurrency atop

robust application architectures. Written by the creators of Absinthe, this book will help you take full advantage of these two groundbreaking technologies. Build your own flexible, high-performance APIs using step-by-step guidance and expert advice you won't find anywhere else.

Bruce Williams and Ben Wilson

(302 pages) ISBN: 9781680502558 $47.95

Software Design X-Rays



Are you working on a codebase where cost overruns, death marches, and heroic fights with legacy code monsters are the norm? Battle these adversaries with novel ways to identify and prioritize technical debt, based on behavioral data from how developers work with code. And that's just for starters. Because good code involves social design, as well as technical design, you can find surprising dependencies between people and code to resolve coordination

bottlenecks among teams. Best of all, the techniques build on behavioral data that you already have: your version-control system. Join the fight for better code!

Adam Tornhill

(274 pages) ISBN: 9781680502725 $45.95

Release It! Second Edition



A single dramatic software failure can cost a company millions of dollars—but can be

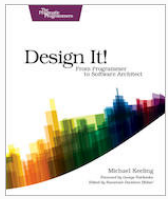
avoided with simple changes to design and architecture. This new edition of the best-selling industry standard shows you how to create systems that run longer, with fewer failures, and recover better when bad things happen. New coverage includes DevOps, microservices, and cloud-native architecture. Stability antipatterns have grown to include systemic problems in large-scale systems. This is a must-have pragmatic guide to engineering for production systems.

Michael Nygard

(376 pages) ISBN:

9781680502398 $47.95

Design It!



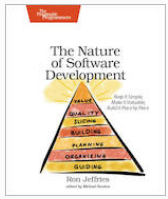
Don’t engineer by coincidence—design it like you mean it! Grounded by fundamentals and filled with practical design methods, this is the perfect introduction to software architecture for programmers who are ready to grow their design skills. Ask the right stakeholders the right questions, explore design options, share your design decisions, and facilitate

collaborative workshops that are fast, effective, and fun. Become a better programmer, leader, and designer. Use your new skills to lead your team in implementing software with the right capabilities—and develop awesome software!

Michael Keeling

(358 pages) ISBN: 9781680502091 $41.95

The Nature of Software Development



You need to get value from

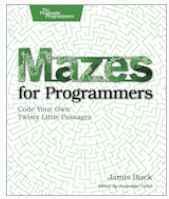
your software project. You need it “free, now, and perfect.” We can’t get you there, but we can help you get to “cheaper, sooner, and better.” This book leads you from the desire for value down to the specific activities that help good Agile projects deliver better software sooner, and at a lower cost. Using simple sketches and a few words, the author invites you to follow his path of learning and understanding from a half century of software development and from his engagement with Agile methods from their very

beginning.

Ron Jeffries

(176 pages) ISBN: 9781941222379 $24

Mazes for Programmers



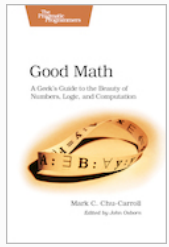
A book on mazes? Seriously? Yes! Not because you spend your day creating mazes, or because you particularly like solving mazes. But because it's fun. Remember when programming used to be fun? This book takes you back to

those days when you were starting to program, and you wanted to make your code do things, draw things, and solve puzzles. It's fun because it lets you explore and grow your code, and reminds you how it feels to just think. Sometimes it feels like you live your life in a maze of twisty little passages, all alike. Now you can code your way out.

Jamis Buck

(286 pages) ISBN: 9781680500554 $38

Good Math



Mathematics is beautiful—and it can be fun and exciting as well as practical. Good Math is your guide to some of the most intriguing topics from two thousand years of mathematics: from Egyptian fractions to Turing machines; from the real meaning of numbers to proof trees, group symmetry, and mechanical computation. If you've ever wondered what lay beyond the proofs you struggled to complete in high school geometry, or what limits

the capabilities of the computer on your desk, this is the book for you.

Mark C. Chu-Carroll

(282 pages) ISBN: 9781937785338 $34